RefactorIT User Manual

Version:	1.9.11
Date:	January 23, 2004

Disclaimer

Aqris endeavors to ensure that the information contained in this manual is correct, but does not accept liability for any error or omission.
RefactorIT is a trademark of Aqris Software AS. All other products mentioned are registered trademarks or trademarks of their respective companies.
Copyright © 2001-2004 Aqris Software AS. All rights reserved.

Credits

Parser built with ANTLR http://www.antlr.org/
Regular expression support built with Jakarta Regexp http://jakarta.apache.org/regexp
JSP support built with Jakarta Jasper http://jakarta.apache.org/tomcat

Introduction

About RefactorIT

RefactorIT is a Refactoring Browser for Java that makes it possible to identify opportunities and apply consecutive refactoring changes safely and intuitively directly from the source code window or the class browser. RefactorIT is designed to be an intuitive, fast, and fun tool to work with - just point the mouse to a context in any navigation or source code windows on the screen, press the right mouse button, and choose from the list of refactoring operations on the context sensitive RefactorIT menu .
Currently in its second release, RefactorIT is being continously improved to further help developers with refactoring and to alleviate developers from the drudgery of manually changing code.
RefactorIT integrates seamlessly into SunTM ONE Studio, Oracle9*i* JDeveloper, JBuilder, and other development environments. RefactorIT can also be installed and used in standalone mode.

About This Manual

We encourage you to send corrections and suggestions to us at support@refactorit.com.

Refactoring

Designing good code from the start of a project is not easy to do.
As a project evolves, it becomes necessary to rethink earlier decisions and change the code accordingly. These changes can result from all issues not having been identified during the initial design or simply from new and changing business requirements . Additionally, even the best designed code starts to decay when developers make quick fixes and workarounds to meet deadlines. This is a natural course of events in most projects. If no appropriate measures are taken, "software rot" starts to occur and, after a while, the whole project becomes a mess. In the end it seems easier to rewrite the project from scratch rather than putting major effort into maintaining the existing software.
This need not be. If the need for change is addressed early on, projects can be kept alive and made more robust and easier to maintain.
Activities such as making code more understandable, re-architecting code, moving code around, removing duplicate code, and so on are collectively known as *refactoring*. The central idea of refactoring is that changes to code do not change functionality: Only changes to code that retain the code's functionality can be called refactoring. Thus refactoring is the process of fighting "software rot" by improving design, readability, and maintainability of code in general. It is not the process of adding new functionality or even significantly changing existing functionality.
When the code is in good shape, functionality can be added or changed with less effort and with fewer bugs. Furthermore, other developers on the project can better understand the code of their colleagues, resulting in increased levels of reuse, fewer bugs, better quality, and faster time to market. The decrease in effort can be quite amazing. The key to good refactoring is to continously perform small refactoring changes. Refactoring should be as natural an activity as changing functionality, implementing new features, testing, and documenting code.

Getting Started

Generally, the source code you intend to refactor should be compilable and RefactorIT must know the source path and class path. However, as RefactorIT is being developed, it is becoming more tolerant of non-compilable source code while still allowing refactoring operations to perform safely.

See Also

Working under Sun ONE Studio
Working in Standalone Version

Concepts

To better understand how to use RefactorIT, it helps to understand the concepts behind the product. For RefactorIT to be able to work properly with your project, you have to define which source and class files belong to the project. You do this by specifying source path and class path respectively.
When RefactorIT loads your project, it parses all source files. If it cannot find types in the project source files, it searches the classpath for them. Note that, unlike the Java compiler, RefactorIT does not require your source files to be 100% compileable. Our goal is to make RefactorIT as tolerant as possible to errors in in source files.

Source Path

Source path lists all .java source files that should be parsed by RefactorIT. Usually, you do not have to list files one by one--you just specify the directory where source files of default (nameless) package are located.
Note that only files belonging to the source path can be modified in the course of refactoring.

Class Path

Class path is similar to the Java CLASSPATH. It lists all directories and archives (jar, zip) to search for compiled classes. Every project has to have at least the J2SE standard classes in the class path. This usually means having rt.jar in the class path.
Note that, as opposed to the Java compiler, RefactorIT ignores the CLASSPATH environment variable. This means that you have to add all libraries your project needs to the class path.

Classes in Source Path vs. in Class Path

Having compiled classes in the class path instead of parsing them from source files speeds up the process of project loading and refactoring operations in general. Memory consumption is also lowered. The primary drawback of this approach is that refactoring operations neither scan nor modify methods of compiled classes. This may, for example, lead to not all usages of your classes being found or, alternatively, some members being erroneously reported as not used. *As a result classes located in source files are first-class citizens and compiled classes are second-class*

citizens with respect to the refactoring process. Thus, you should always have a clear idea of which part of your project should be in the source path and which part should be in the class path in the form of compiled classes.

Running Refactoring Operations

You have many different ways to run refactoring operations or queries supported by RefactorIT. The most straightforward way is to right-click a section of code in the source code window and select the desired action from the popup menu. Note that, similarily, you can right-click packages, classes, methods, and fields in various views of your project in the IDE you use.
You can also run refactoring operations from the Tools->RefactorIT menu of your IDE. If the source window is active, only refactoring-actions for the item under the cursor will be presented. If any other window is active (for instance, a tree-like view of your project), only refactoring operations for the selected item will be presented in the menu. Moreover, you can assign shortcuts for various refactoring operations, to enable you to access them without using the mouse.

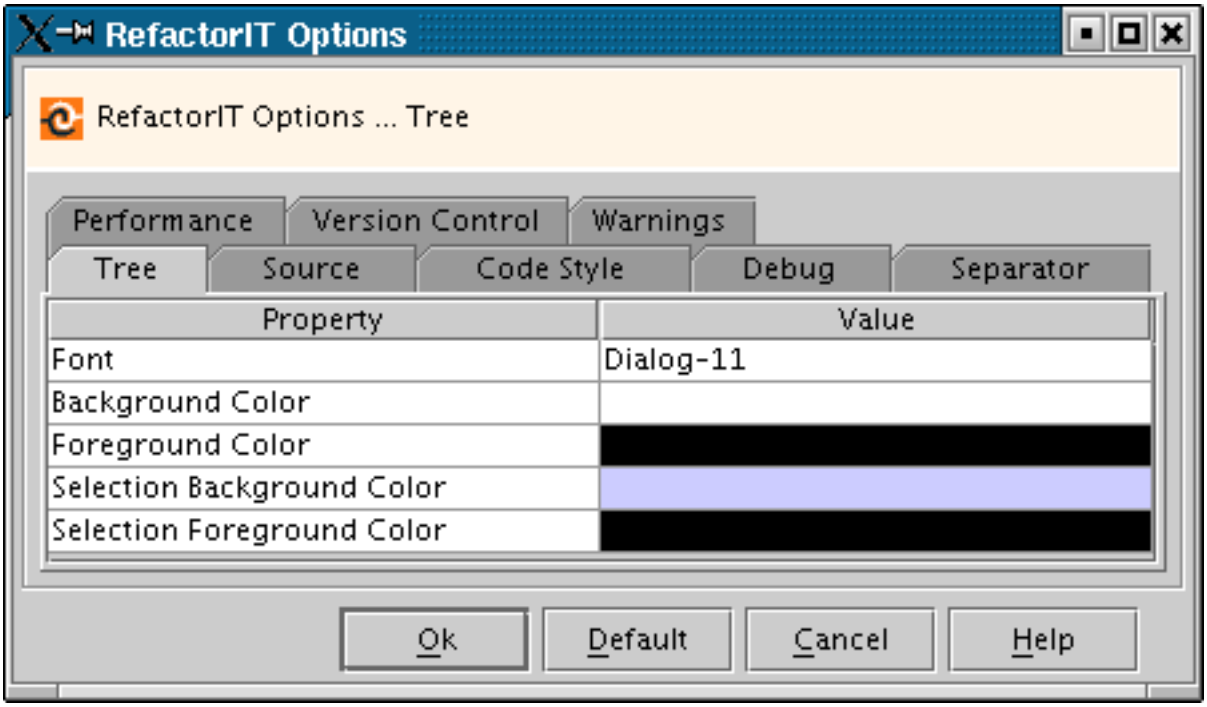
Results of Queries

RefactorIT supports queries like Where Used or Not Used, etc. Most of these queries report results to the output window or "results" panel. The results are presented as a tree. This way you can easily concentrate on a desired subset. Clicking a tree node will open it in the source code window or panel. Right-clicking a node will bring up a list with applicable refactoring operations or queries. From the results panel, you can copy results to the clipboard and paste them into, e.g. a spreadsheet application for more complex analysis.

JSP Files

RefactorIT fully supports JSP (Java Server Pages) in its refactoring operations and queries. If there is .jsp files in your source path you may want to include servlet.jar in your project class path. If RefactorIT is unable to locate servlet.jar it uses servlet.jar in the RefactorIT home directory. The RefactorIT JSP compiler is based on Jakarta's Jasper http://jakarta.apache.org.

RefactorIT Options



This dialog enables you to set global RefactorIT options. If you would like to set project-specific options instead (like source path and class path), please use the RefactorIT Project Options dialog.

Options are saved between RefactorIT sessions. To restore the "factory defaults," use the "Default" button, which restores the default values of the currently open tab.

Tab or setting name	Description
Tree	*Specifies the look of all tree-like views (including folder views and result views for refactoring operations and queries).*
Font	Specifies the font used in the tree nodes
Background Color	Background color of all tree views
Foreground Color	Color of the text used in tree nodes
Selection Background Color	Background color of nodes that have been selected
Selection Foreground Color	Color of text of the nodes that have been selected
Source	*Various options about source code (Except*

	for highlighting, all the settings concerning source code appearance apply only in the standalone browser mode).
Font	The font of source text in standalone browser mode.
Background Color	The background color of the source area in standalone browser mode.
Foreground Color	The color of non-highlighted text in the source area (including regular statements, type names, etc) in standalone browser mode.
Highlight selected line	Whenever RefactorIT takes you to a location in the source code, RefactorIT highlights that line of code if this option is selected. Usually, the highlighting appears in the form of a yellow or blue background color being set for that line (the highlight disappears after some time; it also disappears whenever another line is highlighted).
Highlight Color	Background color of the highlighted line (see the above option for more information about what highlighting means).
Character Encoding	Character encoding used for parsing source files. You can enter a new encoding name that you know is supported in your Java Virtual Machine, or you can select an encoding from the list. Note that RefactorIT does not support Unicode identifiers -- for more information, see the Unicode Support info.
Highlight Color	Background color of the highlighted line (see the above option for more information about what highlighting means).

Code Style	*Various options concerning source code style. These options are used by RefactorIT when it modifies existing code or generates new code.*
Tab Size	Size of TAB character. Common values are 2, 4 or 8.
Use Spaces in Place of Tabs	Whether to replace TAB characters with spaces. Number of spaces is taken from Tab Size.
Block Indent Size	Number of characters that a block of code is indented.
Continuation Indent Size	When an expression is too long to fit onto one line, it is broken into several lines. All lines starting from the second one are indented by using continuation indent.
New Line Before Brace	Whether to place opening curly brace on new line.
Add Space Before Parenthesis	Whether to add space before opening parenthesis.
Separator	*Options for separators used when generating metrics reports.*
Decimal Separator	Specifies the decimal separator used when generating metrics reports.
Grouping Separator	Specifies the thousands separator used when generating metrics reports.
Performance	*These options affect performance.*
Incremental Compiling	When set to true: at rebuild time RefactorIT analyzes class modification dates and rebuilds only those classes that need to be rebuilt.
Call rebuild on every window	Causes project rebuild every time

iconification event	RefactorIT standalone window is iconified.
Use IDE file events (if possible)	Causes RefactorIT to rely on file modification events from IDE, if it possible. Minimizes rebuilding delays when running refactoring on modified code.
Version Control	*Things about file-systems and version control.*
List Of Version Controlled Directories	RefactorIT does not yet have full support for version control, but it can bypass some folders in the source tree that are meant to be used by version control utilities only. For example, it can bypass folders called "CVS", "RCS", etc.
Warnings	*Allows disabling/enabling of various warnings. They can also be disabled from the dialogs that show them.*
Shortcuts	*Shortcuts for standalone browser mode.*

Project Options

The project options dialog box enables you to set project-specific options (like source path and class path, for example). Every platform has its own set of such options:

- Project Options under SunONE Studio
- Project Options under Standalone
- Project Options under JBuilder
- Project Options under JDeveloper

Options are saved between RefactorIT sessions. Each platform has a different file format and location for saving project options.

Cache

RefactorIT maintains an internal cache to save it from having to parse all files in your project each time you run a refactoring operation or query. When source or class files are changed, parts of the cache are invalidated. The next time RefactorIT runs, it brings its cache up to date as efficiently as

it can. Thus, when you make a change to a single source file, only that file will be parsed again. The result is a dramatic improvement in performance most of the time.
However, RefactorIT is not perfect. Sometimes after you make changes to your project, RefactorIT fails to update its internal cache properly. In such situations, RefactorIT can do things like finding deleted classes or reporting that a name is already in use for an element that has been deleted. When you notice such behavior, which can result from the cache's being out of date, you have two tools at your disposal to correct it: Clean and Rebuild. Both tools can be accessed from the RefactorIT menu. The first, Rebuild, has less impact on performance and tries to rebuild outdated parts of the cache. If Rebuild does not help, you can force RefactorIT to purge its cache entirely by running Clean. The next refactoring or query will build the cache from scratch, parsing all source and class files. It might take more time, but will most probably fix anomalies you have experienced because of an outdated cache.

Integration with SunTM ONE Studio and NetBeans

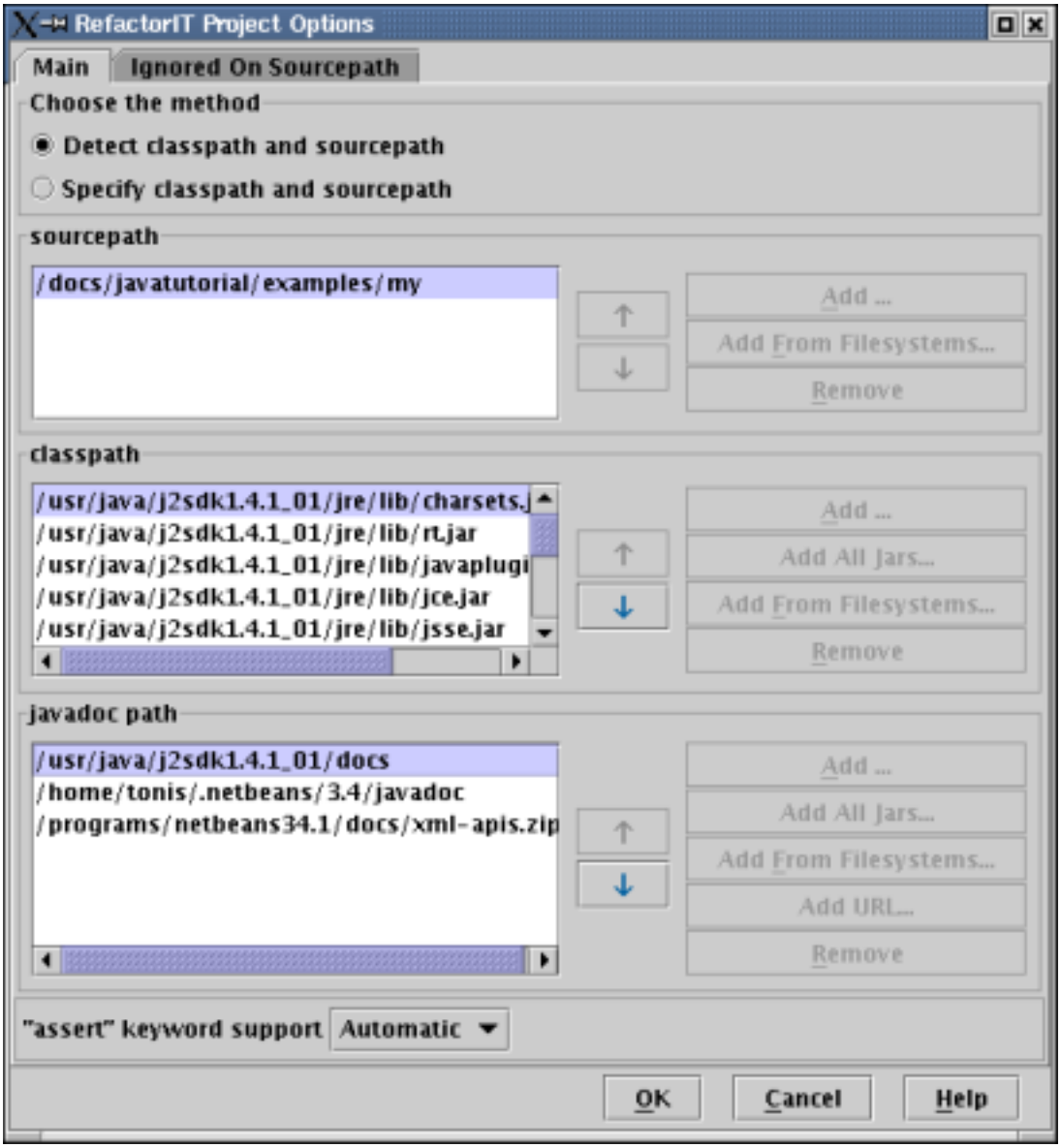
RefactorIT integrates with SunTM ONE Studio (previously known as Forte for Java). To see if RefactorIT has been successfully integrated, open the Tools menu and look for the RefactorIT menu item with submenus.
If you have followed the project definition guidelines of Sun ONE Studio, RefactorIT should be able to detect its project settings automatically, and you do not have to set them separately for RefactorIT. In any case, we strongly recommend that you get familiar with RefactorIT's project settings.
Recently a minor change of behavior has been made (please tell us if you think it needs to be reversed): If you rename a class that is directly linked in the project tab, the link will not be renamed. (The link will still work, however, and this only involves the Rename refactoring.)

See Also

Setting Up a Project
Project Settings
Autodetection of Project Settings
Shortcuts
Version Control System

Project Settings

You can always check and modify project settings in the Project Options dialog. To open the dialog, choose Tools->RefactorIT->Project Options from the IDE main menu (the same command is also available from contextual menus in various places).



You can either let RefactorIT autodetect all the paths, or choose to specify the paths by yourself. (You can read more about autodetection in the section Autodetection Of Project Settings.)

You can also specifiy Assert Keyword mode, with the following results:

- If it is enabled, the Java assert keyword will be supported.
- If it is disabled, the keyword will not be suppoted.
- If it is "automatic" (the default option), RefactorIT will try to guess whether to support the assert keyword or not (it does this by looking at the current JDK version).

From the "Ignored On Sourcepath" tab you can specify packages or filesystems to be ignored that are in the source path. RefactorIT will act as if these items did not exist at all. This is useful, for example, when you need to speed up RefactorIT (or reduce memory requirements) when working with large projects. This can also be useful for (perhaps temporarily) hiding unparseable code from RefactorIT.

Note that ignoring source path items can sometimes also make the rest of the code unparseable by RefactorIT. One way of solving this problem is to put the ignored items in the class path as

compiled class files.

See Also

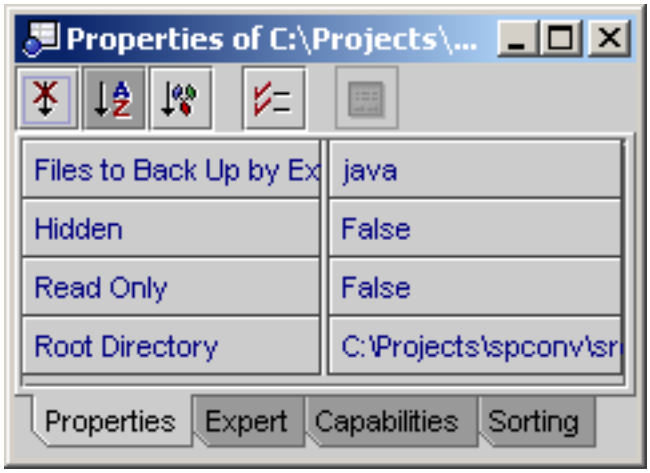
Autodetection of Project Settings

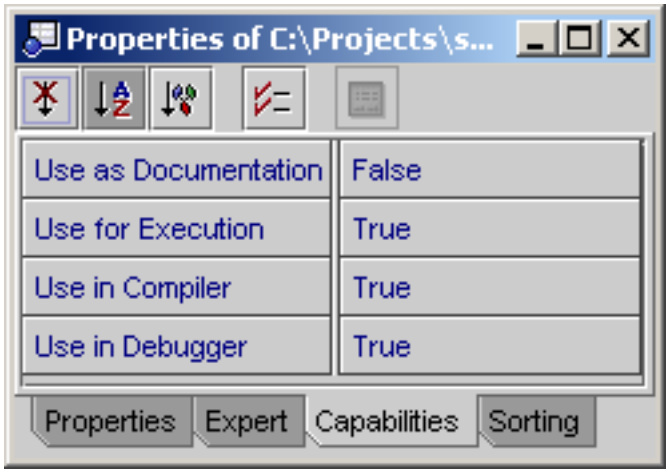
Autodetection of Project Settings

RefactorIT tries to detect the project source path and class path from filesystems mounted in the Filesystems tab of the Explorer window. Thus, if you follow the project mounting guidelines of Sun ONE Studio, RefactorIT will be able to detect your settings properly without any manual intervention.

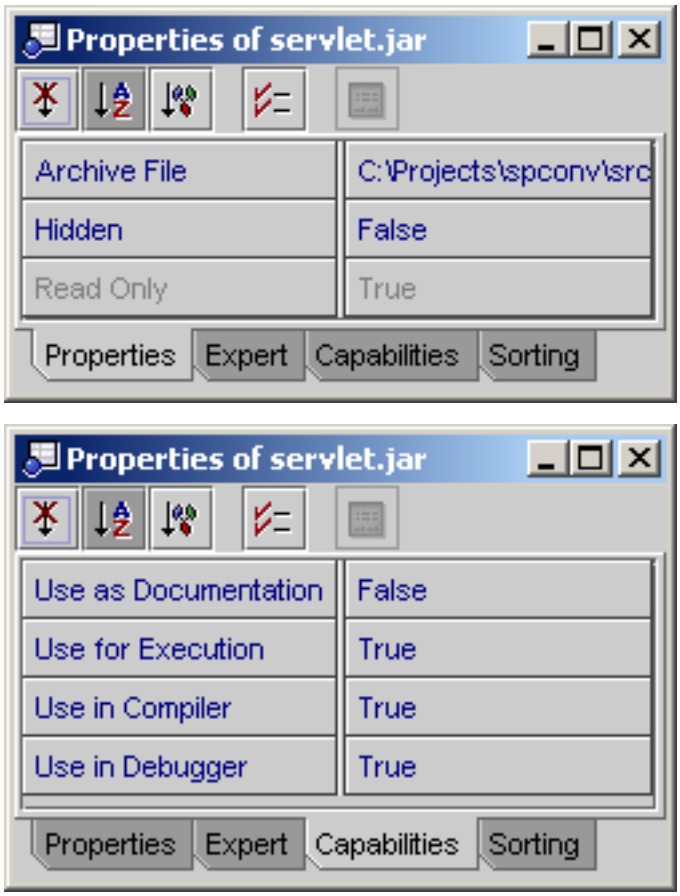
Source path and class path are autodetected as follows:

- Each read-write and visible filesystem marked to be used in the compilation process is added to the source path. The Read Only attribute must be set to False, and Hidden must be set to False in the Properties tab of the filesystem's properties. The Use in Compiler attribute must be set to True in the Capabilities tab of filesystem's properties.





- Each filesystem that is visible and is marked to be used either in the compilation process or for Javadoc comments is added to the class path. The attribute Hidden must be set to False in the Properties tab of the filesystem's properties.The Use in Compiler or Use as Documentation attribute must be set to True in the Capabilities tab of the filesystem's properties.



You can always check to see what source and class path have been detected in the RefactorIT Options dialog.

Setting Up a New Project Under Sun ONE Studio

There are two ways to set up project under Sun ONE Studio: follow guidelines for setting up projects or set up the project in a custom way. We advise the first way as it is ususally easiest. This section describes how to set up your project in the IDE so that RefactorIT will be able to automatically detect its settings. If you do not want to set up the project in the way described below, you can always set the project's settings in the RefactorIT Options dialog (Tools->RefactorIT->RefactorIT Options).
The following project is assumed in the examples below: <project root> + lib - myjar.jar - servlet.jar - ... + java + com + company + product ...

Additionally, it is assumed that your are using Sun ONE Studio 4 or newer.
For Sun ONE Studio to be able to compile your project and for RefactorIT to be able to function, you must properly mount the source files and libraries that use.

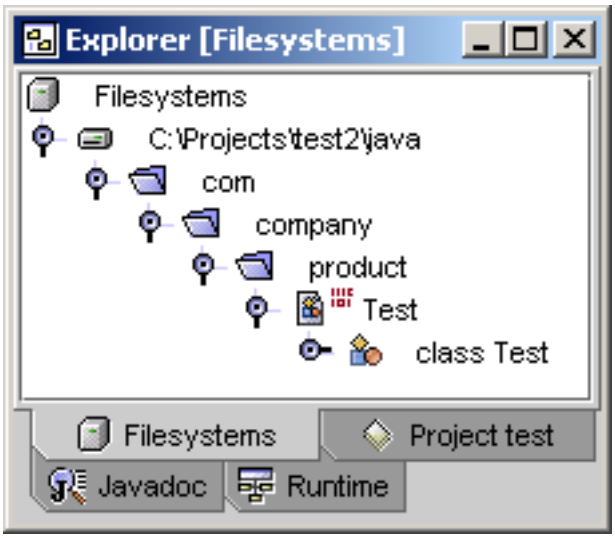
Mounting Source Files

Source files (.java files) must be mounted in order for RefactorIT to function. Mount the default package directory in the Filesystems window. The default package directory is where the default (unnamed) package resides. In this case, it is <project root>/java. The default package directory usually does not contain any .java files. If it does, the source files do not have "package ...;" statement as one of the first rows.

STEPS

1. Go to the Filesystem tab of the Explorer .
2. Right-click the Filesystems node and choose Mount->Local Directory.
3. Choose <project root>/java

A new mounted filesystem should now appear in the Filesystems tab. You can expand it to browse your source files.



Note that if your project uses source files from some other directories, you have to mount those source files as well. Make sure that you always mount directories at the default package root.

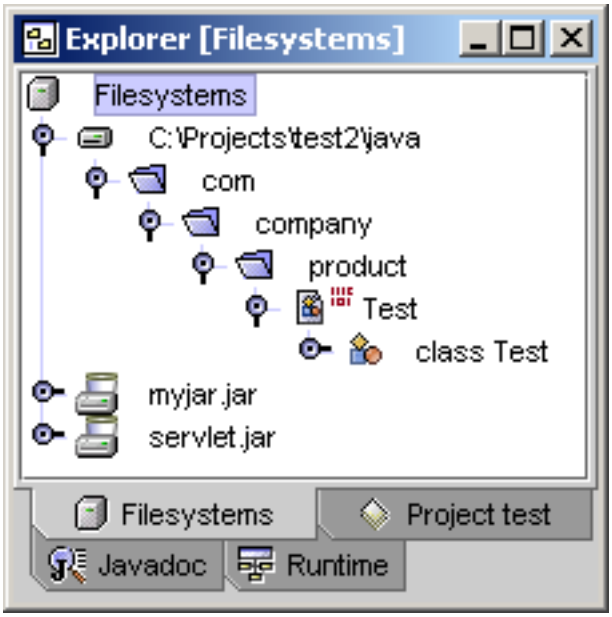
Mounting Libraries

Custom libraries (usually .jar) are required by most projects. They contain compiled classes that your project uses. All libraries required by your project in order to compile need to be mounted. In Sun ONE Studio, any .jar file mounted in the Filesystems tab is automatically treated as such a library and is added to the class path when compiling. In the case of our sample project, all external libraries are located in one directory <project root>/lib.

STEPS

1. Go to the Filesystem tab of the Explorer panel.
2. Right-click the Filesystems node and choose Mount->Archive (JAR, Zip).

3. Choose one of your .jar files (for example, <project root>/lib/servlet.jar). Note that starting from NetBeans 3.3.1 you can add multiple libraries at once.
4. Repeat step 2 and 3 until you've mounted all libraries.



Checking Whether the Project Compiles

If the project as mounted in Sun ONE Studio does not compile sucssefully, RefactorIT might not work properly.

To see if the project compiled:

1. In the main menu, choose Build->Compile All (Ctrl+F9).
2. Check the Output Window for errors.

If your project still does not compile, see what the errors are. Usually, you will be missing an external library. When you find out which library you are missing, mount it and try compiling once again.

It might also be the case that your project does not compile because it uses an external build script (such as Makefile, Ant build.xml). In this case see what that script does to compile your project. Usually, it adds some additional libraries to CLASSPATH. Remember that all of them have to be mounted in the Filesystems tab.

See Also

Mounting Source Files for Reference Only
Mounting Javadoc

Mounting Source Files for Reference Only

Situation: You need to mount some source files but you do not want them to be compiled or used by RefactorIT. For instance, you may want to be able to conveniently access some other project's source files for reference purposes (J2SE source files for example).

Solution: Use the "Ignore On Sourcepath" tab in the RefactorIT Project Options dialog.

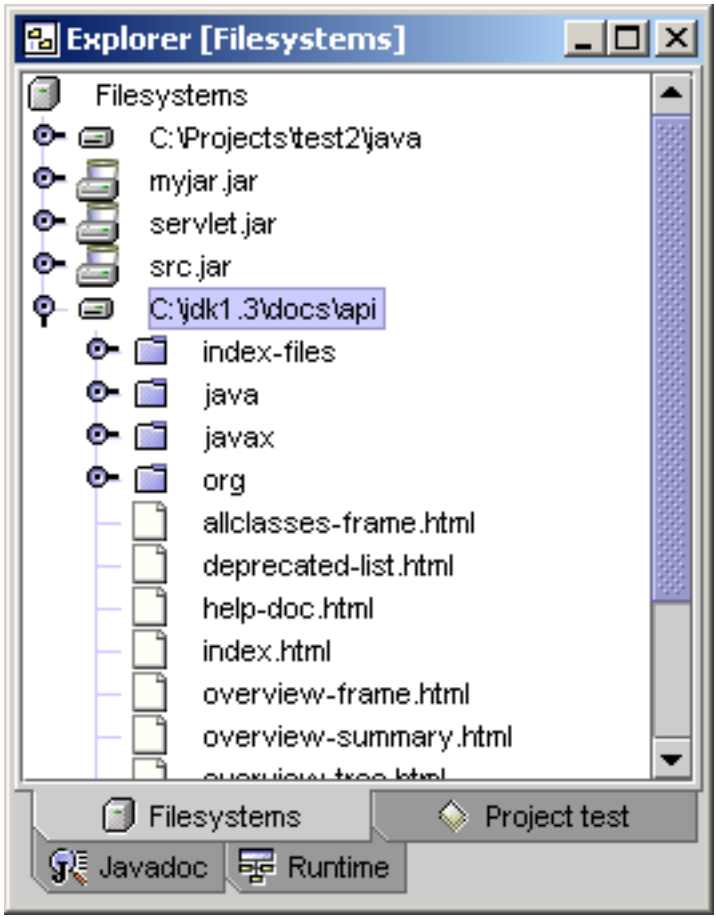
Mounting Javadoc

Situation: You use external libraries (.jar, zip) and have HTML Javadoc for them somewhere. You want SunTM ONE Studio and RefactorIT to show the Javadoc for classes, interfaces, fields, and methods.

Solution: Mount the directories containing the Javadoc and set Use in Documentation to True and all other Use in ... attributes to False in the Capabilities tab of the property sheets of those filesystems:

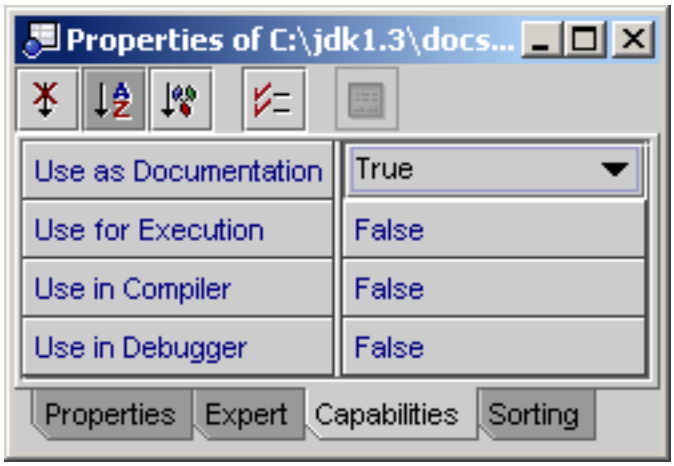
1. Go to the Filesystem tab of the Explorer.
2. docs
   - api
     - index.html
     - index-all.html
     ...

Mount the api directory.



Mount the directory containing the Javadoc. Javadoc usually has the following directory structure:

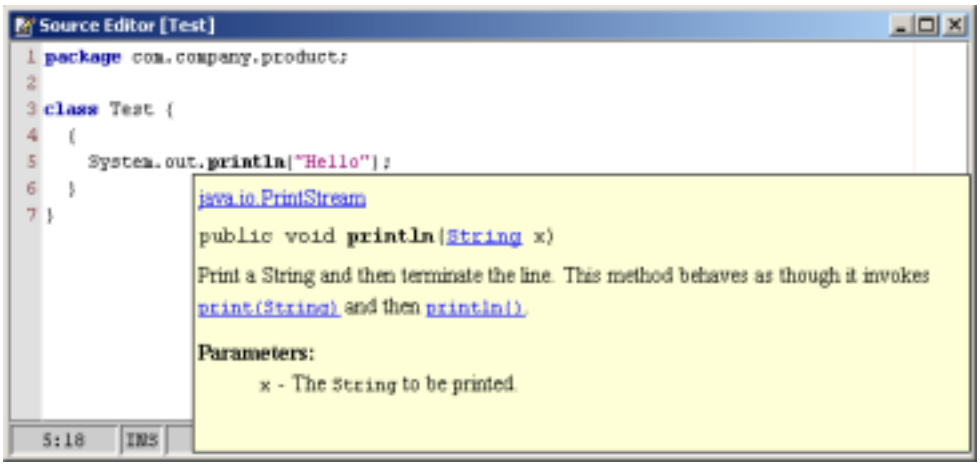
3. Go to the Capabilities tab of the property sheet.



Set Use in Documentation to True and all other Use in ... attributes to False.

Now if you right-click on a class or method in source code, you can do following:

- Choose "Show Javadoc" SunTM ONE Studio's functionality. This will show the Javadoc for any interface, class, method, or field for which you have mounted external HTML Javadoc.
- Choose Tools->RefactorIT->Info. This will show Javadoc for any interface, class, method or field that is declared in your source code or for which you have mounted external HTML Javadoc.

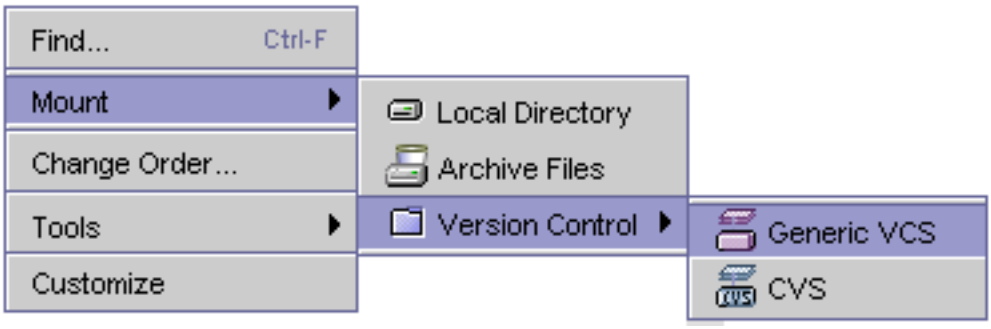


Shortcuts

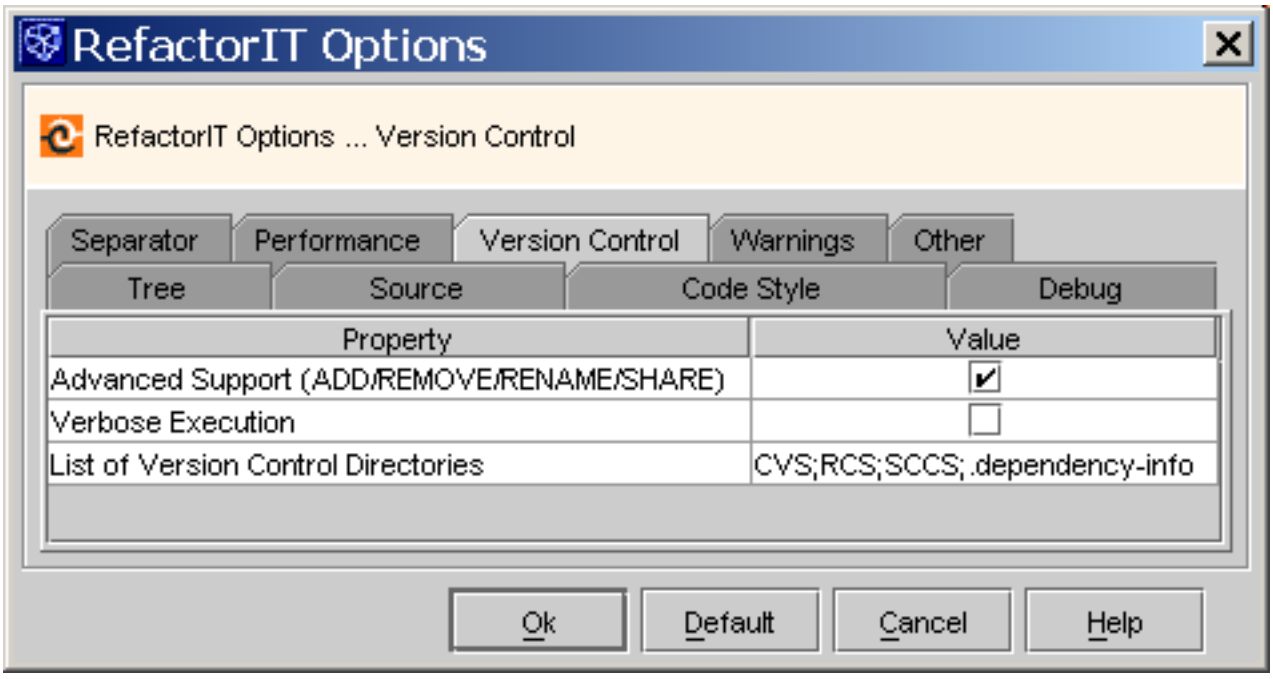
You can configure keyboard shortcuts for RefactorIT actions under Sun ONE Studio. To do that, go to Tools -> Keyboard Shortcuts in the Sun ONE Studio menu. You will see various RefactorIT actions for which shortcuts can be configured.

Version Control System Overview

RefactorIT is capable of issuing VCS specific commands during actions that change file structure. In order to use this capability, the VCS file system should be mounted as a "Generic VCS".



Note: the only **officially supported** VCS system currently is **CVS**, although others might also happen to work.
Please, check that all the commands of the VCS are functional before running any RefactorIT refactoring on a VCS file system. See the Troubleshooting section if you encountered any misbehavior.
You might find it useful to configure specific VCS support options of RefactorIT before first use. Options are located under the RefactorIT Options/Version Control tab.



- **Advanced Support (ADD/REMOVE/RENAME/SHARE)** - With this option disabled, RefactorIT will be able to call GET, CHECKOUT and EDIT commands only when it needs to edit a read-only file or a file that has not yet been received . If you want RefactorIT to support the full range of VCS commands available, you should enable this option. *(enabled by default)*
- **Verbose Execution** - When enabled, all the VCS actions behave the same way as they do when they are invoked manually through the Sun ONE Studio IDE, and

they show configuration dialogs and confirmation messages. When this option is disabled, RefactorIT does its best to keep the operation as silent as possible. During complex operations like Move Class the number of commands called is significant, so it's worth to keep this option disabled unless you need to debug some questionable behavior of RefactorIT. *(disabled by default)*

- **List Of Version Controlled Directories** RefactorIT can bypass some folders in the source tree that are meant to be used by version control utilities only. For example, it can bypass folders called "CVS", "RCS", etc.

CVS specific

- **read-only files** - RefactorIT will issue an EDIT command before editing such files. *NOTE: this behavior of RefactorIT cannot be switched off.*
- **rename** - If you rename a class and its file should be renamed also, the old file will be removed by the REMOVE command and the new file will be added by ADD.
- **move** - If the directory structure changes (e.g. Move Class, Rename Package), directories will be added or removed as needed by the ADD and REMOVE commands.

Committing changes

To keep things as safe as possible RefactorIT doesn't commit the changes made, so that you can always try to recover the previous VCS state if something went wrong or you just changed your mind and don't want to continue with the selected refactoring.

If everything went fine and you agree with the changes made, then invoke the Commit command from NetBeans's menu to commit your changes to your VCS.

See Also

Project Settings
Troubleshooting

Troubleshooting and Limitations

The VCS support of RefactorIT was tested with CVS only, however all the other version control systems presented in Sun ONE Studio as a "Generic VCS" profiles may work well if they behave similarly and provide ordinary CVS-style set of commands.
Please, check that all the commands of the VCS are functional before running any RefactorIT refactoring on a VCS file system.
Since RefactorIT uses the same VCS commands that are accessible through Sun ONE Studio interface, it is assumed that all the VCS commands work properly when invoked manually within the Sun ONE Studio.

It was noticed that if your server version is newer that the version of the cvs client Sun ONE Studio was configured to use (including internal cvs-client), then REMOVE invoked on a file sometimes results in "cvs [server aborted]: no such directory '\<name_of_parent_directory\>'". In this case you can try to use another cvs client, for example command line client *cvs.exe* of the WinCvs.

If REMOVE fails for CVS...

The complete log of all the VCS commands called by RefactorIT together with accompanying information is saved into the *ide.log* file, which is located under the system directory of your Sun ONE Studio user home directory.

Integration with JBuilder

RefactorIT integrates well with JBuilder starting with version 5.

Note: RefactorIT has Productivity! compatibility mode which could be useful if you have Productivity! and RefactorIT integrated on the same JBuilder installation.

To check if RefactorIT has been successfully integrated, look for RefactorIT in the main menu. The RefactorIT menu should contain several submenu items. In addtion, you should see a number of RefactorIT buttons in the main toolbar.

RefactorIT takes its sourcepath and classpath from the JBuilder Project Settings.

See Also

Toolbar buttons
Results Pane
Project Settings
Shortcuts
Productivity! Compatibility Mode

RefactorIT Toolbar Buttons

There are five RefactorIT toolbar buttons in main toolbar. You can modify the toolbar display by checking or unchecking selections on the View|Toolbars menu. RefactorIT provides toolbar buttons for the following commands:

Where Used

Rename

Info

GoTo Declaration

Back Action (Returns your cursor position back after *"GoTo Declaration"* was performed)
You can use these toolbar buttons in all views of JBuilder. Just select or set caret on desired unit and click any toolbar button.

RefactorIT Operation Results Pane

A tabbed result pane appears at the bottom of the JBuilder AppBrowser for displaying result tree or table of performed RefactorIT operation.
RefactorIT generates a new tab at the bottom of the message pane for each performed RefactorIT operation. These tabs allow you to interact with the source view.
You can also hide the result pane by right-clicking the message tabs and choosing Hide Message View, or pressing Ctrl+Alt+M. Close individual tabs or all tabs in the result pane by pressing the Close Tab/Close All button.
You can also copy from the result pane. Right-click the desired pane and select the text you want to copy and select Copy To Clipboard. Some RefactorIT operations provide toolbar buttons on the tab that let you restart the current process or customize it.

Project specific settings

Use the JBuilder Project Properties dialog box to set the RefactorIT project specific properties for the active project. To open it, choose Project|Project Properties, and then select the RefactorIT tab. There you can find following settings:
"assert keyword support" - if it is "enabled", then the Java assert keyword will be supported, if it is "disabled" then the keyword will not be suppoted, and if it is "automatic" (the default option) then RefactorIT will try to guess wether to support the assert keyword or not (it does this by looking at the current JDK version).
You can also change sourcepath and classpath settings for RefactorIT in this dialog. You can either let RefactorIT autodetect all the pahts (the default option), or choose to specify the paths by yourself.
Important: In order to work with a project most effectively, understand how JBuilder uses paths and libraries. Check the list of references in JBuilder help for more information.

Shortcuts

You can use the Keymap Editor to view and change individual keybindings for the RefactorIT operations. The Keymap Editor is viewable in all editions of JBuilder, and customizable in JBuilder SE and Enterprise. Access the Keymap Editor in order to view or change keymappings in either of two ways: Choose Tools|IDE Options. Select the Browser tab. Click Customize. Or, Choose

Tools|Editor Options. Select the Editor tab. Click Customize. To learn how to use the Keymap Editor, click the Help button in the Keymap Editor dialog.

Productivity! Compatiblity Mode for RefactorIT

Productivity! is a JBuilder plugin from AMIS Software (http://www.softamis.com/).
When run normally, Productivity! and RefactorIT can interfere with each other, usually resulting in inaccessible functionality (for example, see http://www.refactorit.com/bugzilla/show_bug.cgi?id=1682).
When RefactorIT is run in this compatibility mode then the tools will interfere much less with each other, at the expense of removing RefactorIT menu items from the JBuilder structuew view.
This mode can be turned on or off from RefactorIT tab in the JBuilder Project Properties dialog (you will need to restart JBuilder for the settings to take effect).

Integration with Oracle JDeveloper

RefactorIT integrates with Oracle9*i* and Oracle10*g* JDeveloper. To check if RefactorIT has been successfully integrated, look for RefactorIT in the main menu. The RefactorIT menu should contain several submenu items. In addtion, you should see a number of RefactorIT buttons in the main toolbar.
RefactorIT uses JDeveloper Project Settings to find out which source files are part of the project and to resolve all classes referenced in the project. For this reason, it is of importance that the source paths of the JDeveloper project are correctly configured. Especially, that any source path is the root of the package tree and that all external classes used in the source files are available in the project libraries. The general rule of thumb is that if you are able to compile the project under JDeveloper, then also RefactorIT can handle it.

See Also

Toolbar buttons
Results Pane
Project Settings

RefactorIT toolbar buttons

There are five RefactorIT toolbar buttons in the main toolbar. The buttons invoke the following commands:

Back Action (Returns your cursor position back after *"GoTo Declaration"* was performed)

Where Used

Rename

Info

GoTo Declaration

RefactorIT Results pane

A tabbed result pane appears at the bottom of the JDeveloper main window for displaying results of the RefactorIT analysis and smart query functions.
RefactorIT generates a new tab at the bottom of the message pane for each performed RefactorIT operation. These tabs allow you to interact with the source view.
You can close individual results by clicking on the black cross button on the results pane toolbar. The button with a red cross closes all results of a particular search (e.g. Where Used). You can close all results by clicking on the close buuton in the right-top corner of the results pane.
You can also copy from the result pane. Right-click the desired pane and select the text you want to copy and select Copy To Clipboard.
Some RefactorIT operations provide toolbar buttons on the tab that let you restart the current process or customize it.

Project specific settings

Use the JDeveloper Project Settings dialog box to set the RefactorIT project specific properties for the active project. To open it, choose Project|Project Settings. In the RefactorIT section of the settings dialog, find following setting:
"assert keyword support" - if it is "enabled", then the Java assert keyword will be supported, if it is "disabled" then the keyword will not be suppoted, and if it is "automatic" (the default option) then RefactorIT will try to guess wether to support the assert keyword or not (it does this by looking at the current JDK version).

Shortcuts

You can assign keyboard accelerators to all RefactorIT operations. To configure key bindings, select Tools|Preferences... from the JDeveloper main menu. In the Accelerators section of the shown dialog, choose RefactorIT category to see all available RefactorIT actions.

Setting Up a New Project In Standalone Version

The first time you run the standalone version, you will have to create a new project. The next time you run RefactorIT, you can choose to open an existing project or to create another new project. To create a new project you have to do the following:

- Click the Add button in the class path section and select the libraries of your project (jar, zip). Note that selecting the directory where the libraries are located is not the correct approach. Instead, add the libraries themselves that are contained in the directory.
- Click Add all JARs in the class path section and select a folder containing your libraries. All libraries from that folder will be added at once.

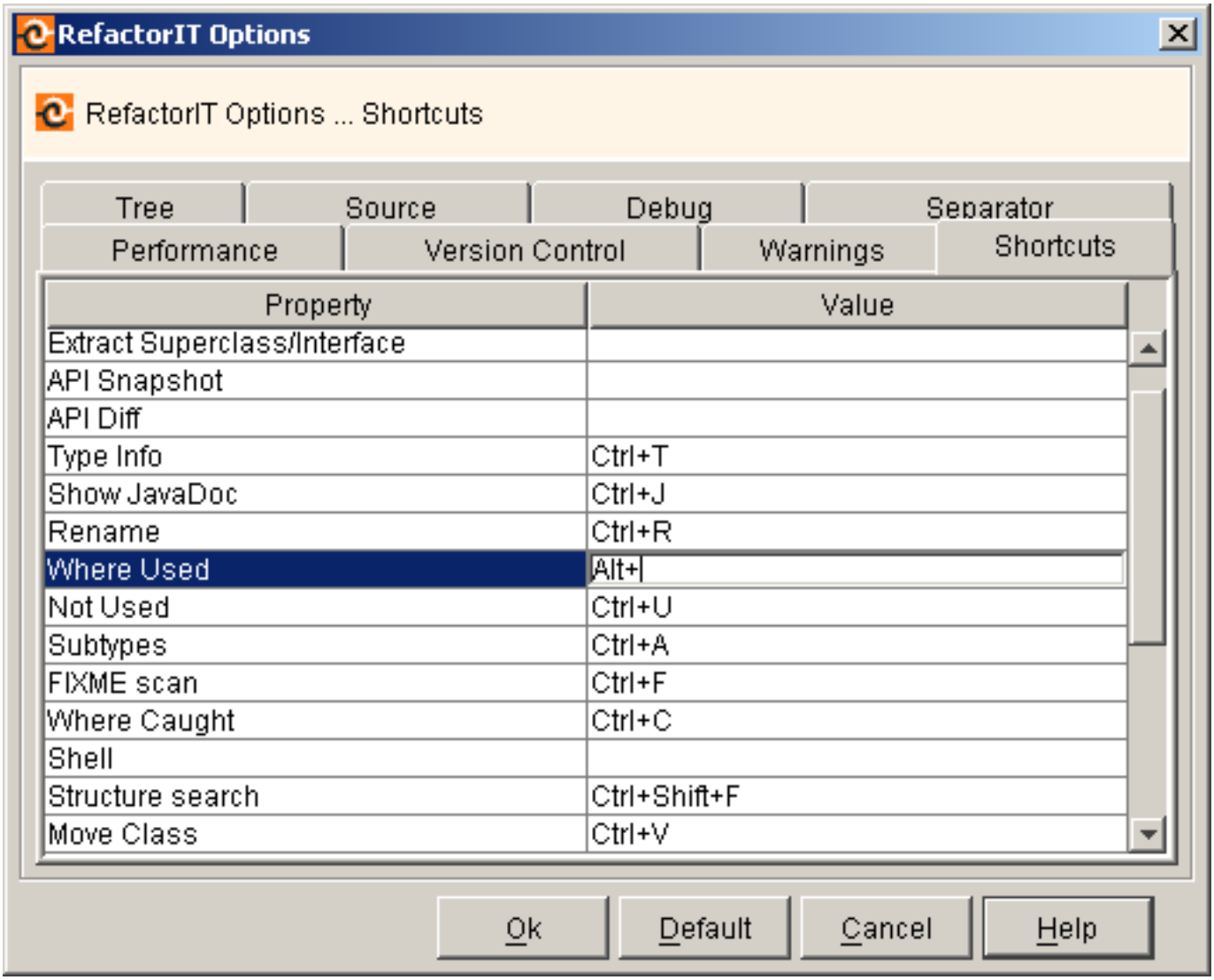
Click the Finish button to open your new project.

1. Click the Save As button.
2. Select a target folder to save settings of the project to.
3. Give a name to your project.
4. Click Save and Next.
5. Specify the source path of your project. Click Add in the source path section and select a folder containing your project files (.java). If your project uses source files from several directories, you have to add all source files. Do not forget to always add the directories at the default package root.
6. Specify the class path of your project. There are two ways to do it:

It might take from several a seconds to several minutes depending on project size to load the project. If some source files are compileable and all libraries have been added, you should see a package tree in the left panel. If parsing finishes with errors, it usually means that a library is missing or some source code is not compilable.

Shortcuts

You can modify the default shortcuts through the Shortcuts Options dialog: Choose View->RefactorIT Options, and select the Shortcuts tab.



>

Double-click a cell in the right column of the shortcuts table to start editing.

Once you're in the cell editing mode, type in the keystroke combination you want to use. For instance, to make a function map to Ctrl + Shift + d, hold down the Ctrl key and the Shift key, then press d. When you have set the key bindings to suit your needs, click another cell in this table to apply the changed keybindings immediately.

If you press Bacspace or keep this cell empty, the keystroke sequence is unbound from the selected action and it will be removed from the list.

Note: You can use your shortcuts everywhere in Standalone mode except in the source view.

Menu activation - F10

File menu - Alt+F

View menu - Alt+V

Project menu - Alt+P

Help menu - Alt+H

File menu functions:

New project - Ctrl+N
Open existing project Ctrl+O
Save existing project Ctrl+S
Exit RefactorIT program Ctrl+X

Project menu functions

Rebuild project - Alt+R

Help menu functions

Start help - Alt+F1

RefactorIT operations

Where Used - Ctrl+W
Rename - Ctrl+R
Not used - Ctrl+U
Metrics - Ctrl+E
Go to Declaration - Ctrl+B
Fixme Scan - Ctrl+F
Subtypes - Ctrl+A
Type Info - Ctrl+T
Show JavaDoc - Ctrl+J
Move Class - Ctrl+V
Show Dependencies - Ctrl+D
Call Tree - Ctrl+L

Refactorings Overview

RefactorIT provides the following queries and refactoring operations:

- Audit
- API Snapshot / API Diff
- Call Tree
- Change Method Signature
- Clean Imports
- Convert Temp To Field
- Create Constructor
- Create Factory Method
- Create Missing Method
- Cross-referenced HTML
- Encapsulate Field
- Extract Method
- Extract Superclass/Interface
- Fixme Scan
- Go to Declaration
- Inline Method
- Inline Variable
- Introduce Explaining Variable
- Metrics
- Minimize Access Rights
- Move Method/Field
- Move Class
- Not Used
- Pull Up / Push Down
- Rename
- Show Dependencies
- Structure Search
- Subtypes
- Type Info
- Where Caught
- Where Used

Note: RefactorIT saves all modified files before it runs any refactoring operation or query.

See Also

Undoing Refactoring Operations
Version Control Support
Results Panel
Unicode Support

Command Line Interface and Ant task

Results Panel

Results of queries are shown in the Results Panel.
The left part of the panel features this toolbar:

browse (move back and forward between) query results.

view and modify settings of this panel.

close query result

close entire tab of query results (not present under newer versions of SunTM ONE Studio -- these versions already have buttons for that)

show or hide hidden rows (there is a hide/unhide command in popup menu of result rows)

show context-sensitive on-line help.

refresh results --reruns the active query after you've made some changes

filter results of the active query.

Undo/Redo

Since version 2.0 RefactorIT has full undo support for all refactorings. Undo will revert all files modified by the refactoring. If user has changed the files RefactorIT will ask before continuing. User can also Redo every Undo. Redo will revert files to the exactly state they were before last Undo. So if user applied refactoring, modified files and reverted refactoring using Undo then Redo will restore all these modifications.

Version Control Support

Currently we support VCS only under NetBeans and Sun ONE Studio, as described in their Version Control System Overview. We are planning to support more versions in the future. Please vote at support@refactorit.com if you need some specific VCS support under some specific IDE.

Unicode Support

RefactorIT does have Unicode support.
You might need to change character encoding in the RefactorIT Options dialog, because RefactorIT does not import source encoding settings from IDEs it runs under.
Please write to support@refactorit.com if you find any problems with some special characters.

Add Delegate Methods

Creates delegate methods to the field of the target class. To execute action click on class or class field name and select methods to create for the field.

API Snapshot / API Diff

API Snapshot saves the current state of the API. API Diff compares API snapshots, enabling you to see the changes that have been made to the API.
API Snapshot contains signatures of methods, constructors, fields, classes, interfaces, and packages. Included are items with all access modifiers: public, private, protected, and package private.
To create an API Snapshot, select the packages and classes that you'd like to include in the snapshot. (You can also include the whole project by selecting the root of the filesystem.) Then excute the snapshot command from the pop-up menu of the selected items.
To compare the current state of the code against an existing snapshot, select the code as you did when creating the snapshot, and then select API Diff from the pop-up menu.

Audit

Audit warns about suspicious or useless code (for example, assigning a variable to itself, local variables that have the same names as fields, and unused import statements). For some of these warnings there are automated fixes (for example, Clean Imports for the Unused imports warning). The results can be sorted by priority, density of results per file, warning type, and location. To switch between sort modes, just click the appropriate column headers in the results table.
Audit could be run on separate files or packages or on entire projects. Each time you run it you are shown a filter dialog, making it possible to turn various warnings on or off. The main options include:

- Unused local vars
- Unused imports
- Variables shading fields
- Empty blocks
- Method parameter modifications
- Self-assignments

Sorting Results

Audit results can be sorted by priority, density of results per file, warning type, and location. To switch between sort modes, click the appropriate column headers in the results table. (By default results are sorted by their location.)
Every warning has a different *priority*: for instance, warnings about unused imports have low priority, whereas self-assignment has a high priority.
Density of results is currently calculated with the following algorithm:

```
d = 0
foreach warning in file
  d = d + priorityCode(warning)
d = d / lineCount(file)
```

The priorityCode in the above formula is read form the refactorit.cfg file (from properties named audit.violation-priority-score.*xxxx*). The default values are: 10 for low priority, 20 for normal priority, and 50 for high priority.

Unused Local Vars

This warning reports unused local variables (optionally including method and catch clause parameters). No automatic fixes are available.

Unused Imports

This warning reports import statements that could be safely removed from the code without breaking it. Often such import statements are a result of doing a lot of manual refactoring.

Automatic fixes (Clean Imports) are available from a pop-up menu.

Variables Shading Fields

This warning reports local variables whose names match the names of fields in the same class (optionally including constructors' and setters' parameters). An automatic fix is available form the pop-up menu (renaming the local variable).

Empty blocks

This warning reports empty blocks such as empty ' if and while statements:

```
if(something); // Possibly an error
  doSomething();
```

To surpress this warning, you can either disable it from the filter dialog or you can replace the suspicious semicolon with {}. No automatic fixes are available for this warning.

Method Parameter Modifications

This warning reports modifications of method parameters such as :

```
public void aMethod(int parameter) {
  parameter = 0; // A modification here
  System.out.println(parameter);
}
```

No automatic fixes are available for this warning.

Self-assignments

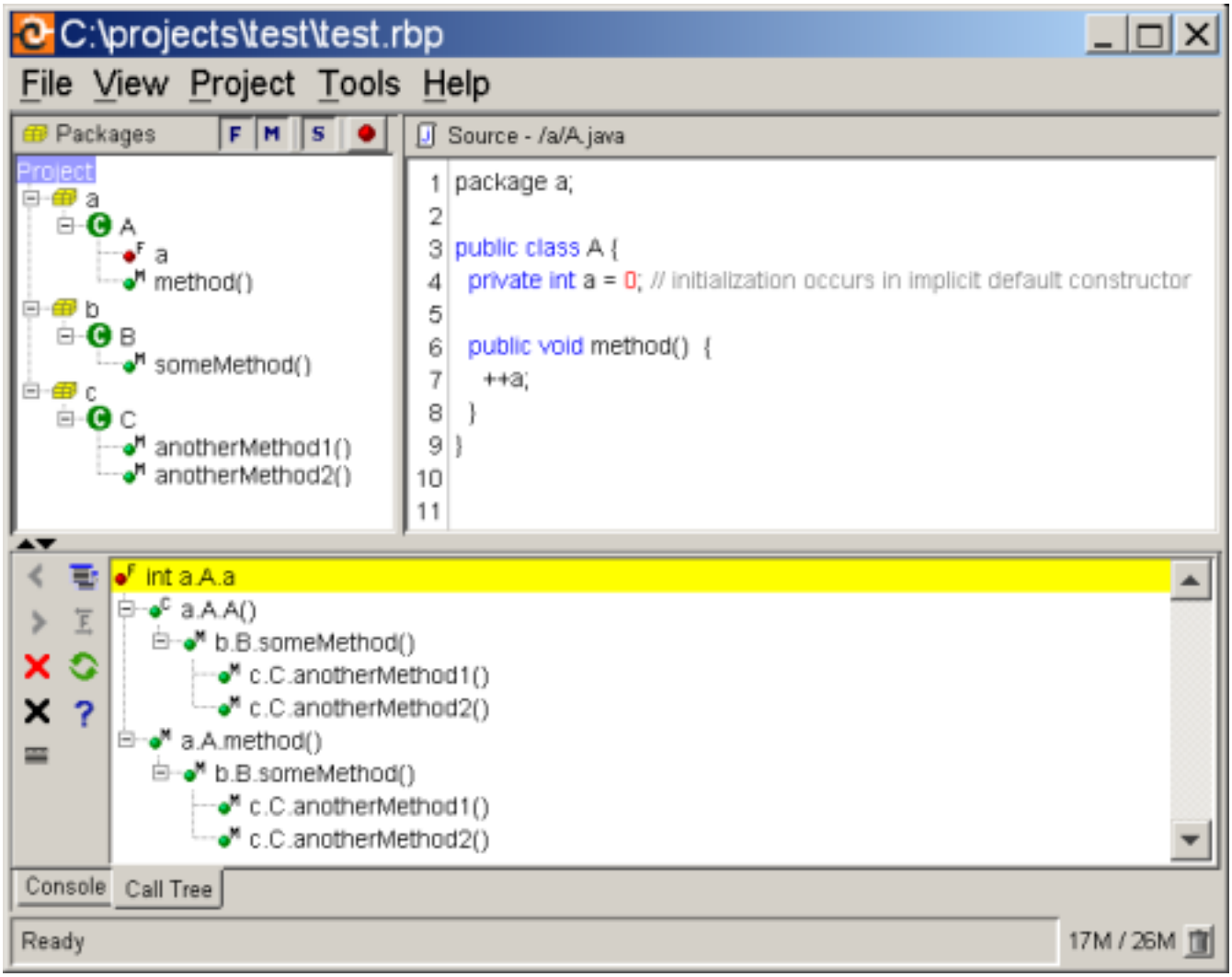
This warning reports assignments for which the left and right side are the same:

```
  ...
  int x;
  boolean b;
  ...
  x = x; // Error
  ...
  if(b = b) // Error
  ...
```

No automatic fixes available for this warning.

Call Tree

Shows all possible stack traces for a given method, constructor or field.
Result looks like a recursive Where Used ran in an 'implementation' mode on all results found until there is no more occurencies found.
Hint: sometimes can be quite useful to see all the ways some field can be accessed or changed during real runtime of the program.



See Also

Where Used
Where Caught has a similar technology, but works for exceptions

Clean Imports

Clean Imports removes unneccessary import statements. You can invoke this command on the entire project or on packages or files independently. You can then review the changes and switch them on or off before saving them.

Create Constructor

Create Constructor enables you to easily create constructor definitions from existing class variables. You select the class variables and run Create Constructor refactoring, and the constructor is created from those selected declarations. Create Constructor refactoring also works for inner classes.
RefactorIT will report when creation of a constructor is impossible or might break the existing project.
RefactorIT does its best to preserve the layout of your code. Still, there are some cases where your coding conventions might be violated as a result of running Create Constructor. Thus, you must review each constructor after creating it.

See Also

Create Constructor Example

Example

The sample class below just encapsulates the Person's properties.

```
/**
 * Encapsulates the Person's main properties.
 */
public class Person {
  // Person properties
  private String name;
  private Date birth;
}
```

Select the field variables:

```
  // Person properties
  private String name;
  private Date birth;
```

Run Create Constructor by right-clicking on the selection and choosing RefactorIT->Create Constructor. A new construcor will be created:

```
 /**
  * @param name
```

```
   * @param birth
   */
  public Person(String name, Date birth) {
    this.name = name;
    this.birth = birth;
  }
```

The sample class will now look like this:

```
/**
 * Encapsulates the Person's main properties.
 */
public class Person {
  // Person properties
  private String name;
  private Date birth;

  /**
   * @param name
   * @param birth
   */
  public Person(String name, Date birth) {
    this.name = name;
    this.birth = birth;
  }
}
```

Create Factory Method

Create Factory Method enables you to create factory methods from existing constructors. The primary rationale is that when there are many constructors with similar parameters, it is better to provide clients with factory methods that have descriptive names. Moreover, sometimes complex construction logic is required that cannot be accomplished in a standard constructor.
Create Factory Method creates a static method with the same signature as a constructor. The method returns a newly created instance of the class and Create Factory Method does the tedious task of delegating all parameters to the constructor for you. You are then free to modify the new method to suit your needs. For a code example, see example.

See Also

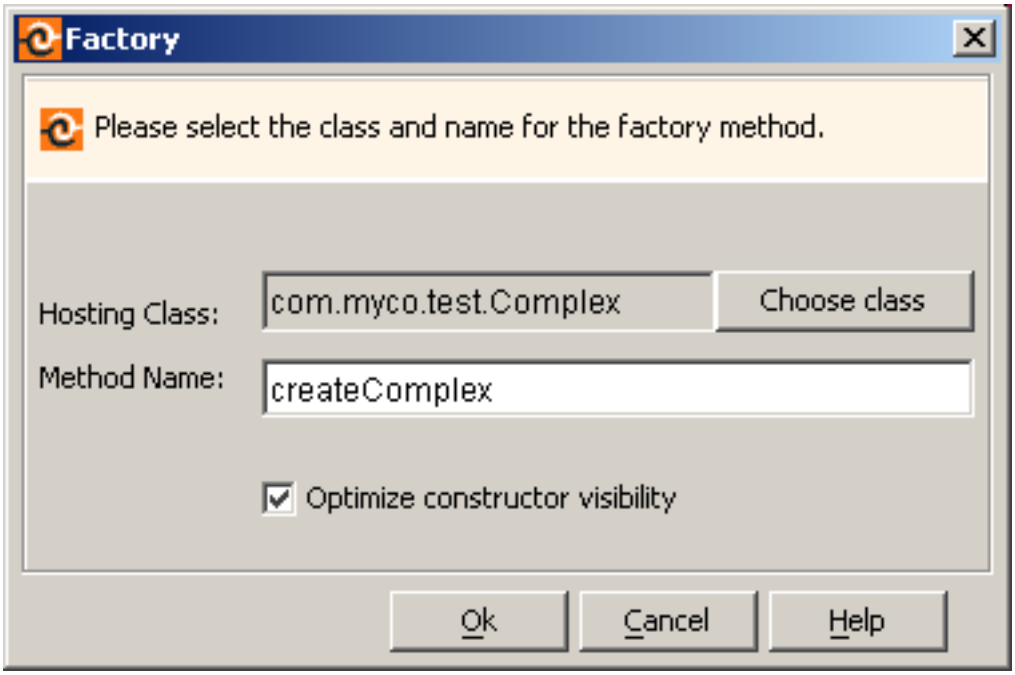
Example
Replace Constructor with Factory Method in Martin Fowler's online catalog of refactoring operations

Create Factory Method Example

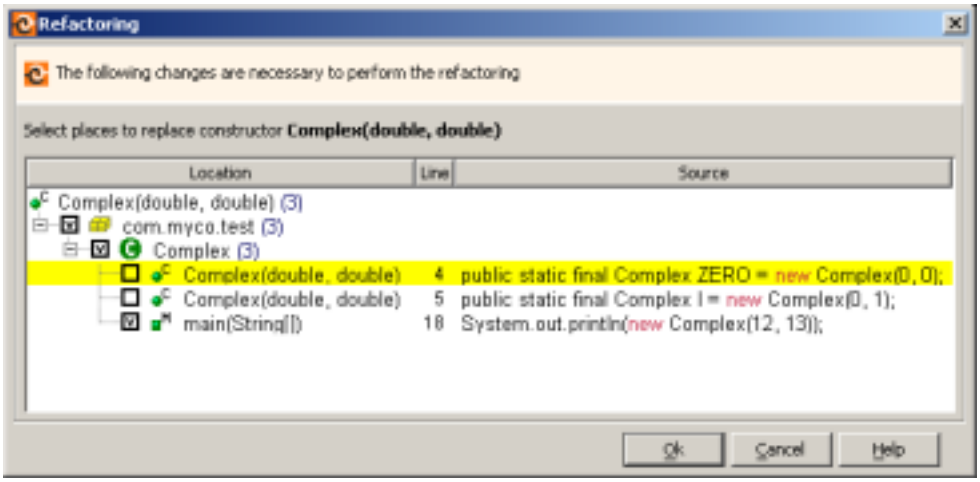
In this example we start with a class that represents complex numbers.

```
public class Complex {
  ...

  public static final Complex ZERO =
    new Complex(0, 0);
  public static final Complex I =
    new Complex(0, 1);

  /**
   * Creates new complex number with
   * specified real and imaginary parts.
   *
   * @param real  real part.
   * @param imag  imaginary part.
   */
  public Complex(double real, double imag) {
    ...
  }

  public static void main(String[] args) {
    System.out.println(new Complex(12, 13));
  }
}
```

The constructor is intuitive. However, we expect that in many cases numbers representing 0, i, -i, and so on will be created. We would like to create only few instances of these common numbers. As a result, the constructor does not suit us well, since a new instance is always created.
To create a factory method, click the constructor and run Create Factory Method. Next, specify valueOf as the name of the method, and then click OK to create the new method.
[*Note: The following dialog uses the name createComplex and not valueOf.]*



RefactorIT then shows you all the places in the source path that invoke the constructor. By default, RefactorIT will replace all these invocations with invocations of the new factory method. In our case, we don't want instantiation of constants (ZERO and I) to be replaced because we don't want to complicate the algorithm. We do, however, replace the constructor invocation in method main.



After you click OK, RefactorIT generates the new method valueOf. Additionally, the invocation of constructor is replaced by a call to the new method inside the main method, as shown in the following code:

```
public class Complex {
  ...

  public static final Complex ZERO =
    new Complex(0, 0);
  public static final Complex I =
    new Complex(0, 1);
```

```
  /**
   * Creates new complex number with
   * specified real and imaginary parts.
   *
   * @param real  real part.
   * @param imag  imaginary part.
   */
  public Complex(double real, double imag) {
    ...
  }

  public static Complex valueOf(double real,
                                double imag)
  {
    return new Complex(real, imag);
  }

  public static void main(String[] params) {
    System.out.println(valueOf(12, 13));
  }
}
```

Next, we add code to valueOf that returns the existing instance for zero and i:

```
public static Complex valueOf(double real,
                              double imag)
{
  if (real == 0) {
    if (imag == 0) {
      return ZERO;
    } else if (imag == 1) {
      return I;
    }
  }

  // custom number
  return new Complex(real, imag);
}
```

A final touch is to make the constructor private, so that clients cannot invoke it directly and have to use valueOf instead:

```
private Complex(double real, double imag) {
  ...
```

```
}
```

The refactored code now looks as follows:

```
public class Complex {

  public static final Complex ZERO =
    new Complex(0, 0);
  public static final Complex I =
    new Complex(0, 1);

  /**
   * Creates new complex number with
   * specified real and imaginary parts.
   *
   * @param real  real part.
   * @param imag  imaginary part.
   */
  private Complex(double real, double imag) {
    ...
  }

  public static Complex valueOf(double real,
                                double imag)
  {
    if (real == 0) {
      if (imag == 0) {
        return ZERO;
      } else if (imag == 1) {
        return I;
      }
    }

    // custom number
    return new Complex(real, imag);
  }

  public static void main(String[] params) {
    System.out.println(valueOf(12, 13));
  }
}
```

To summarize, Create Factory Method can be used to simplify the creation of factory methods that delegate to constructors. You are spared the tedious task of delegating parameters, which is especially handy when a constructor has many parameters. Another helpful feature is that you can let Create Factory Method replace invocations of the constructor with calls to the new method.

Create Missing Method

Creates a method that is called but does not exist. There are two possibilties to invoke it. You can click on method invocation or select &ldquoCreate Missing Method&rdquo item from the pop-up menu in the RefactorIT Errors tab. (First you have to get RefactorIT to display the source parsing error, and then you can right-click on the error).
One way of forcing RefactorIT display the Errors tab is to invoke a refactoring command (like WhereUsed, which you might need to invoke more than once). Invoking Clean and Rebuild might not show parsing errors.

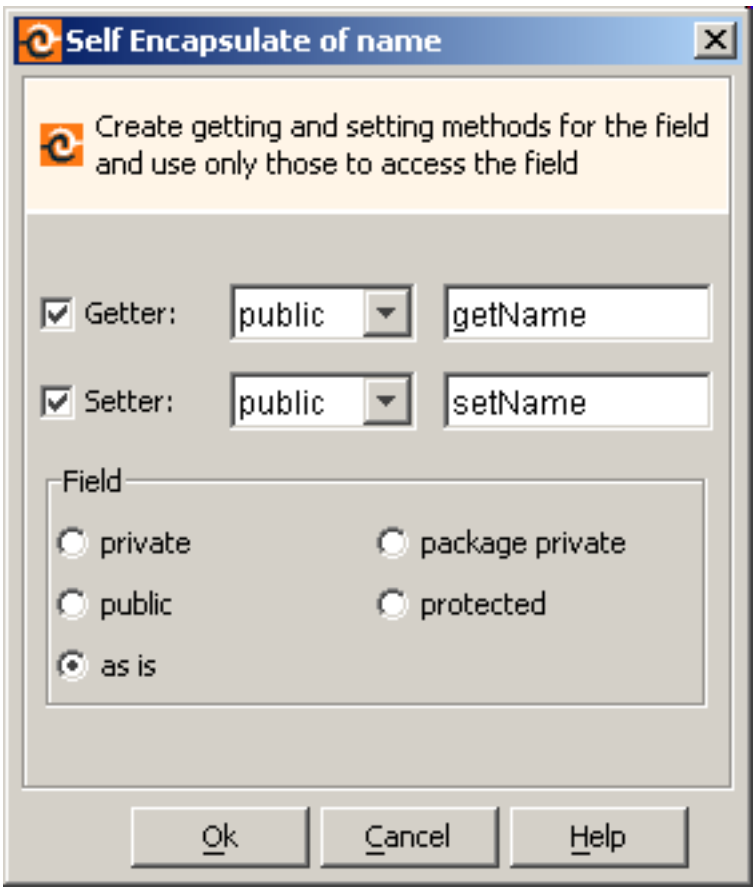
Cross-Referenced HTML

Creates a project's source code listing in HTML with cross-referencing links between items. (For example, you can click on a method invocation in the listing to get to the source of the invoked method.)
This command is useful for quick browsing of the source code when you are learning how the code works.
Cross-Referenced HTML creates one HTML file for each source file. All the created files are named after fully qualified source file names (for example, com.mycom.MyClass.html).
This command is available from different menus depending on how you are running RefactoriIT:

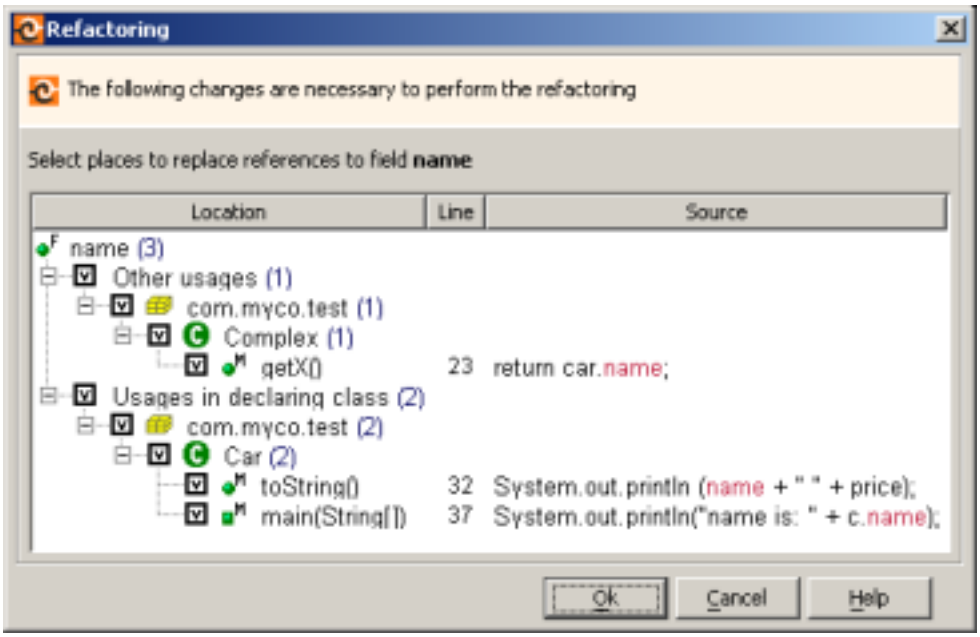
- With RefactorIT running in standalone mode, this command is available from the Project menu.
- Under JDeveloper and JBuilder, this command is available from the RefactorIT main menu.
- Under SunOne Studio, this command is available from the Tools->RefactorIT menu.

Encapsulate Field

Encapsulate Field creates getters and setters for a field. It is additionally capable of replacing usages of the field by invocations of the new getters and setters.
To use this command, click the desired field and run Encapsulate Field, then choose names for the getter and setter. In addition, you can change visibility of the field itself. For example, you can make the field private, thus forcing clients to use getters and setters. When you are done, click OK to continue with encapsulation.



RefactorIT lists all usages of the field in the source path. You can choose which occurrences get replaced by calls to the getter or setter and which ones remain unchanged.
When you are done with the following dialog, click OK to complete encapsulation.



See Also

Encapsulate Field and Self-Encapsulate Field in Martin Fowler's online catalog of refactoring operations

Extract Method

Extract Method enables you to break large methods into smaller ones. You select a section of code and run Extract Method. The code is replaced with a call to a newly created method containing that section of code.
If extracting the selected section of code is impossible or might break your project, RefactorIT will report potential problems and ask for confirmation.
RefactorIT does its best to preserve the layout of your code. Still, there can be cases where your coding conventions might be violated as a result of Extract Method. Therefore, it is recommended that you review the extracted method afterwards.

See Also

Extract Method Example
Extract Method in Martin Fowler's online catalog of refactoring operations

Example

The code below validates name according to several rules.

```
void validateName(String name, List errors) {
  if (name == null) {
    throw new NullPointerException("name");
  }

  if (errors == null) {
    throw new NullPointerException("errors");
  }

  // Check length
  if ((name.length() < 3) ||
      (name.length() > 10))
  {
    errors.add("Invalid name length");
  }

  // Check that name consists
  // only small letters or digits
  final char[] nameChars = name.toCharArray();
  for (int i = 0, len = nameChars.length;
       i < len; i++)
  {
    final char c = nameChars[i];
```

```
    if (!(((c >= '0') && (c <= '9'))
          || ((c >= 'a') && (c <= 'z')))) {
      errors.add(
        "Name can only contain [a-z0-9]");
      break;
    }
  }
}
```

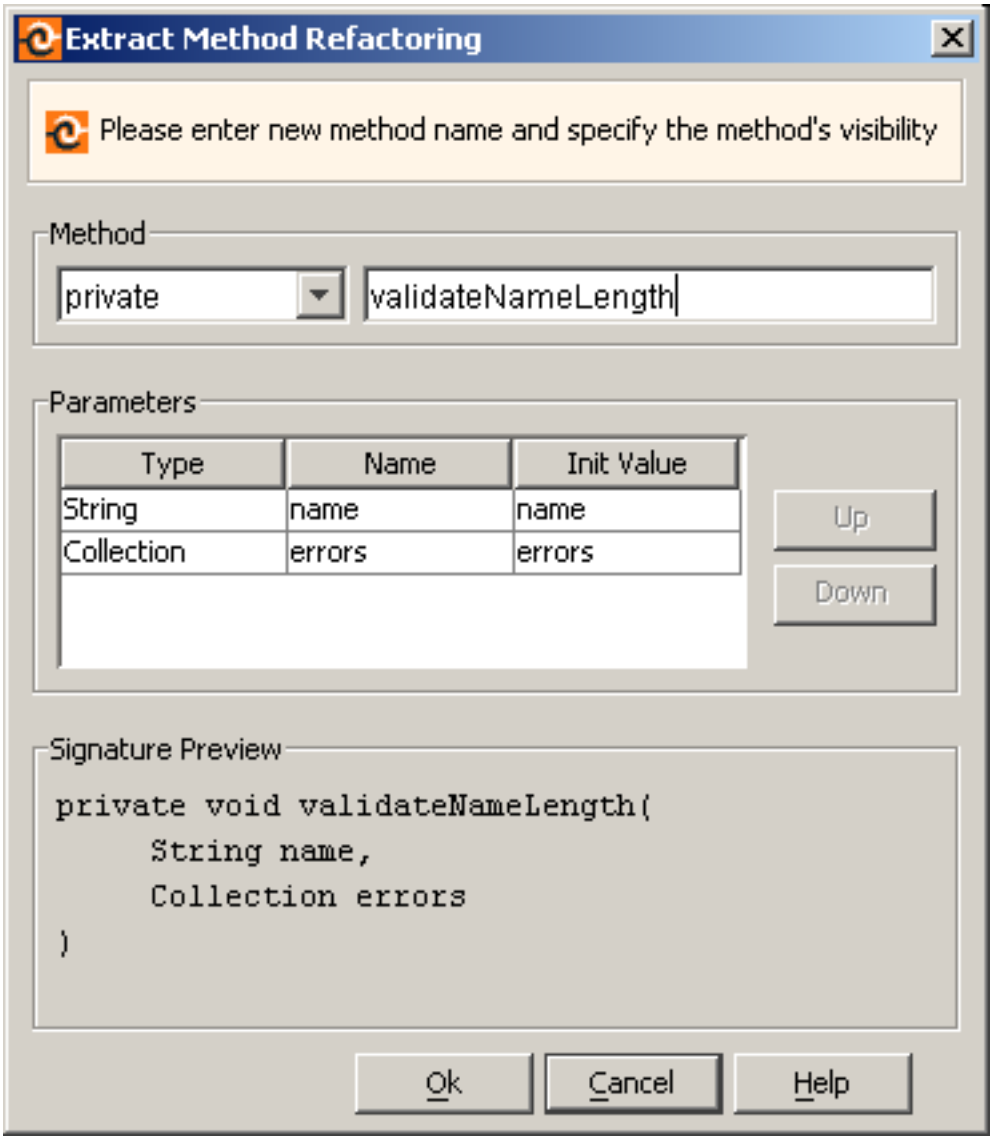
You receive a request to add a new method. There are two good candidates: a section of code that checks whether length is acceptable and another section of code that checks whether name consists only of small letters and digits. You select the code that performs each function and use Extract Method to produce the two new methods.

Select the three lines that perform length validation:

Extracting Length Validation

```
 if ((name.length() < 3) ||
     (name.length() > 10))
 {
   errors.add("Invalid name length");
 }
```

Run Extract Method by right-clicking on the selection and choosing RefactorIT->Extract Method. Pick validateNameLength as the name of the new method.



The new method is created as follows:

```
private void validateNameLength(String name,
                                List errors)
{
  if ((name.length() < 3) ||
      (name.length() > 10))
  {
    errors.add("Invalid name length");
  }
}
```

The original code now contains a call to the extracted method like the following:

```
void validateName(String name, List errors) {
  if (name == null) {
    throw new NullPointerException("name");
  }
```

```
  if (errors == null) {
    throw new NullPointerException("errors");
  }

  // Check length
  validateNameLength(name, errors);

  // Check that name consists
  // only small letters or digits
  final char[] nameChars = name.toCharArray();
  for (int i = 0, len = nameChars.length;
       i < len; i++)
  {
    final char c = nameChars[i];

    if (!(((c >= '0') && (c <= '9'))
          || ((c >= 'a') && (c <= 'z')))) {
      errors.add(
        "Name can only contain [a-z0-9]");
      break;
    }
  }
}
```

In the same way as with the previous extraction, select the code that validates whether name contains only allowed characters. Run Extract Method and specify validateNameCharacters as the name of the extracted method.

Extracting Validation of Characters Allowed

The new method is created as follows:

```
private void validateNameCharacters(
  String name,
  List errors
){
  final char[] nameChars = name.toCharArray();
  for (int i = 0, len = nameChars.length;
       i < len; i++)
  {
    final char c = nameChars[i];

    if (!(((c >= '0') && (c <= '9'))
```

```
          || ((c >= 'a') && (c <= 'z')))) {
        errors.add(
          "Name can only contain [a-z0-9]");
        break;
      }
    }
}
```

The original code now contains a call to the extracted method like the following:

```
void validateName(String name, List errors) {
  if (name == null) {
    throw new NullPointerException("name");
  }

  if (errors == null) {
    throw new NullPointerException("errors");
  }

  // Check length
  validateNameLength(name, errors);

  // Check that name consists
  // only small letters or digits
  validateNameCharacters(name, errors);
}
```

Now just remove superfluous comments and the refactored code will look like this:

The Result

```
void validateName(String name, List errors) {
  if (name == null) {
    throw new NullPointerException("name");
  }

  if (errors == null) {
    throw new NullPointerException("errors");
  }

  validateNameLength(name, errors);
  validateNameCharacters(name, errors);
}
```

```
private void validateNameLength(String name,
                                List errors)
{
  if ((name.length() < 3) ||
      (name.length() > 10))
  {
    errors.add("Invalid name length");
  }
}

private void validateNameCharacters(
  String name,
  List errors
){
  final char[] nameChars = name.toCharArray();
  for (int i = 0, len = nameChars.length;
       i < len; i++)
  {
    final char c = nameChars[i];

    if (!(((c >= '0') && (c <= '9'))
          || ((c >= 'a') && (c <= 'z')))) {
      errors.add(
        "Name can only contain [a-z0-9]");
      break;
    }
  }
}
```

Extract Super

Extract Super enables you to extract methods and fields into a new superclass or an interface.

See Also

Extract Superclass
Extract Interface

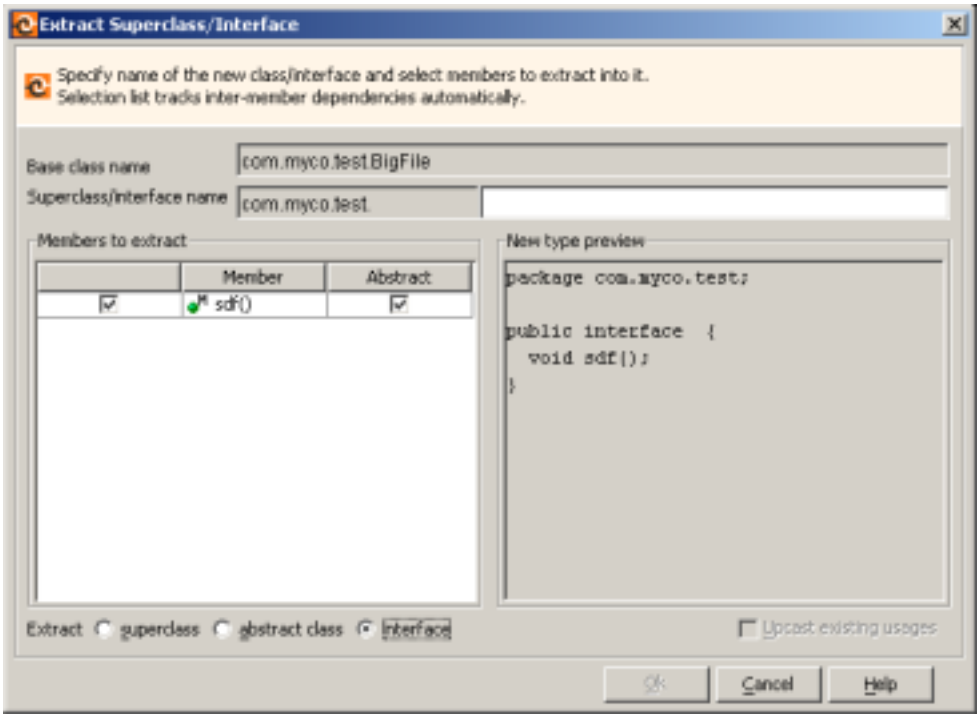
Extract Interface

Extract Interface enables you to extract methods and static constants from a class into a new interface. The original class will implement the new interface.

Only public instance methods can be extracted. Fields must be declared as public static final in order to be extracted.

To extract an interface:

1. Select a class.
2. Run Extract Superclass/Interface.
3. Select the Extract Interface radio button.
4. From the list of fields and methods, select the ones to be extracted to the new superclass.
5. Click OK to extract the new superclass.



See Also

Extract Superclass

Extract Interface in Martin Fowler's online catalog of refactoring operations

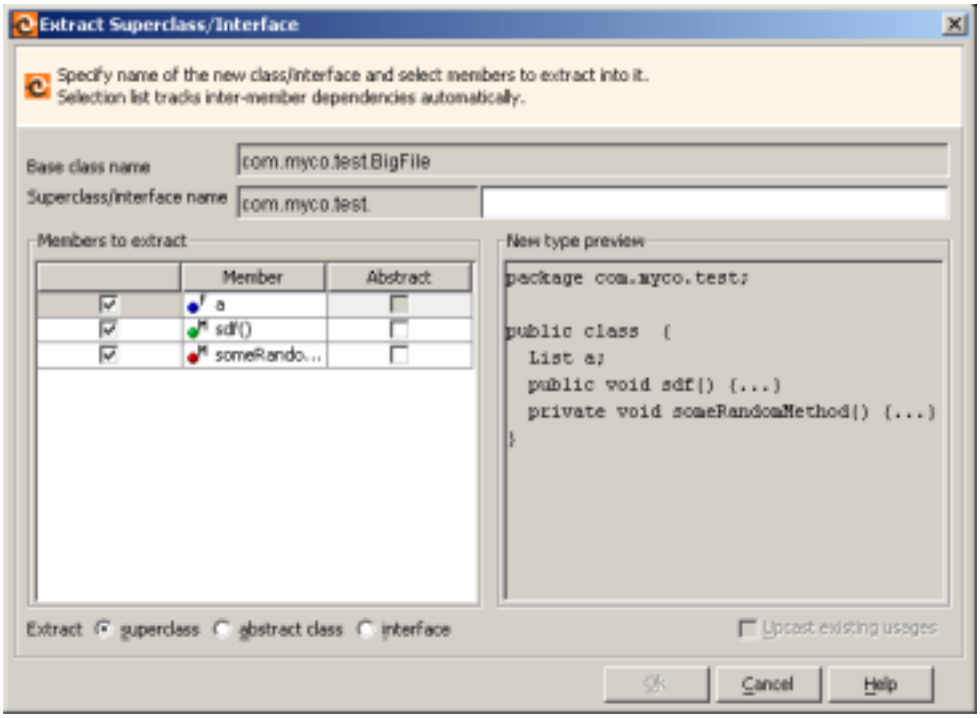
Extract Superclass

Extract Superclass allows you to extract methods and fields of a class into a new superclass. The original class will extend the new superclass.

Private instance methods and fields cannot be extracted into the abstract superclass.

To extract a superclass:

1. Select a class.
2. Run Extract Superclass/Interface.
3. Select the Extract Superclass radio button.
4. From the list of fields and methods, select the ones to be extracted to the new superclass.
5. Click OK to extract the new superclass.



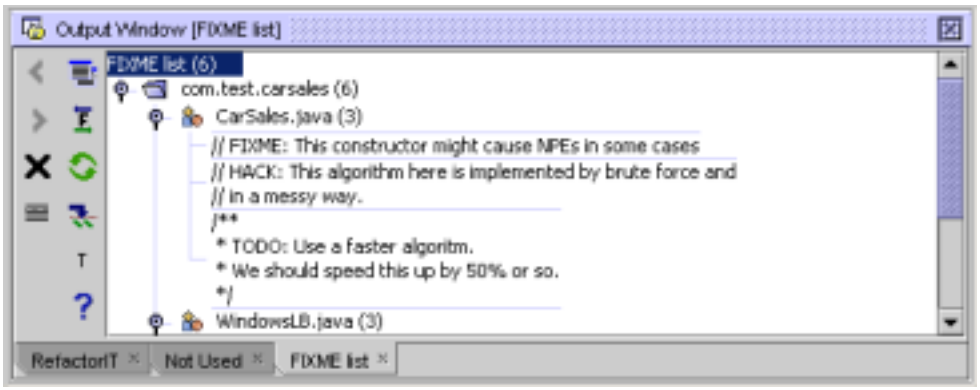
See Also

Extract Interface

Fixme Scan

Fixme Scan is a powerful and configurable search tool for FIXME-like comments in the source code. It also enables you to add timestamps to your FIXME comments in order to track their history.

You can run Fixme Scan for a type, a package, or an entire project. The results of the search are shown in an output window structured like a package tree. Double-clicking an entry takes you to the comment's location in the source window.


Output Window [FIXME list]
FIXME list (6)
com.test.carsales (6)
CarSales.java (3)
// FIXME: This constructor might cause NPEs in some cases
// HACK: This algorithm here is implemented by brute force and
// in a messy way.
/**
* TODO: Use a faster algoritm.
* We should speed this up by 50% or so.
*/
WindowsLB.java (3)
RefactorIT
Not Used
FIXME list

See Also

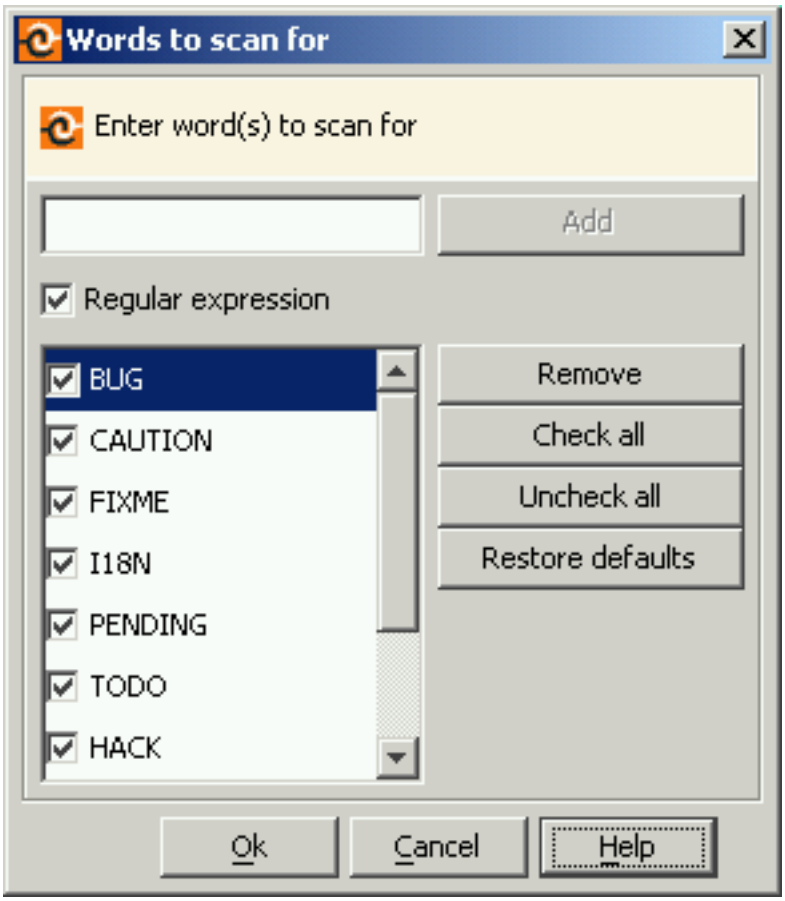
Words Scanned

Timestamps

Words Scanned

In the Words Scanned dialog box, you can configure the rules that the Fixme Scanner uses to detect FIXMEs . This dialog box is displayed every time you run the FIXME Scanner. If the dialog box is not open, click the Words button (

) on the left side of the output window.
The following graphic shows the Words Scanned dialog box:



You can use the checkboxes on the left to choose which words are included in the current search. The words (or rules in general) are run against every line in a comment to see if there is at least one match. There are two kinds of rules in the list: regular expressions and line prefixes. (Regular expression is the default type.)
For example, the following regular expression locates all comments that contain "FIXME" and "John" (in that order):

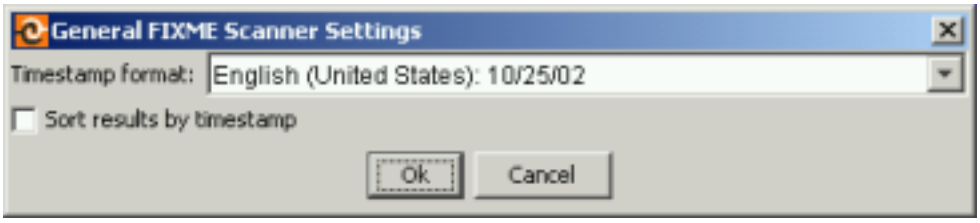
```
FIXME.*John.
```

For more information about the format of these expressions, see the API documentation of Jakarta-Regexp.

Timestamps

If you want to timestamp your FIXME-like comments in source code, press the T button on the left side of the FIXME Scanner output window. This button adds today's date to all comments in the current output window (if they don't already have a timestamp in a recognized format).
Note: Selection in the result panel has no effect here--all comments in the result window are processed by the timestamper, no matter which ones are or are not selected.
You can also sort results by timestamps. To do so, press the Filter button (

) on left side of the output window, select the checkbox in the dialog that appears, and click OK.



Go To Declaration

Goes to the variable, class or method declaration point if it's on the source path.
Go To Type Declaration goes to the variable type declaration.
Go To Super Declaration goes to the overriden method declaration point for method and superclass or superinterface declaration point for type.

Inline Method

Put the method's body into the body of its callers.
You can run Inline Method on a name in method declaration or on a name in method invocation.

See Also

From Martin Fowler's online catalog of refactoring operations: Inline Method

Inline Variable

Replaces all references to a variable with the expression in its declaration and removes the declaration.
Inline variable can be ran on fields (with some restrictions) and local variables.

See Also

From Martin Fowler's online catalog of refactoring operations: Inline Temp

Introduce Explaining Variable

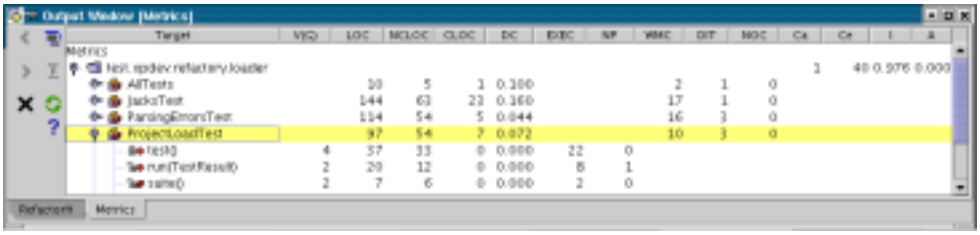
Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.
You can run Introduce Explaining Variable on selection of single or complex expression in the source panel.

See Also

From Martin Fowler's online catalog of refactoring operations: Introduce Explaining Variable

Metrics

Metrics calculates various metrics for projects, packages, types, members, and constructors.



The following table shows the metrics that are available and provides a description of each metric:

Name	Description
LOC, SLOC	Total Lines Of Code
NCLOC, NCSL, ELOC	Non-Comment Lines Of Code
CLOC	Comment Lines Of Code
DC	Density of Comments: DC = CLOC / LOC
NP	Number of Parameters in method's signature
EXEC	Executable statements
V(G), CC	McCabe's Cyclomatic Complexity (see the V(G) link at left)
WMC	Weighted Methods per Class. Sum of V(G) of declared methods of a class
RFC	Response For Class. Number of distinct methods and constructors invoked by a class

DIT	Depth in Tree. Distance from class to root of inheritance hierarchy. 0 for java.lang.Object
NOC	Number of Children--number of direct subclasses of a class
NOT	Number of Types. Number of classes and interfaces
NOTa	Number of Abstract Types. Number of abstract classes and interfaces
NOTc	Number of Concrete Types. Number of concrete classes
NOTe	Number of Exported Types. Number of classes and interfaces exported outside a package
Ca	Afferent Coupling (aka Incoming Dependencies). Number of types outside a package that depend on types of the package.
Ce	Efferent Coupling (aka Outgoing Dependencies). Number of types outside a package that types of the package depend on.
I	Instability (Robert C. Martin). I = Ce / (Ca + Ce)
A	Abstractness. Ratio of abstract classes and interfaces in a package.
D	Distance from the main sequence. Ideal is D = 0. The perpendicular distance of a package from the idealized line A + I = 1.
CYC	CYClic dependencies (momentary for packages only). Number of other packages that the package depends on, and which in turn also depend on the package.

Total Lines of Code (LOC)

Based on http://jakarta.apach e.org/turbine/maven/metrics.html
Metrics derived from lines of code are perhaps the simplest of all the metrics to define and compute. Counting lines has a long history as a software metric dating from before the rise of structured programming, and it is still in widespread use today. The size of a method or a class affects the ease with which it can be understood, its re-usability and its maintainability. There are a variety of ways that the size can be calculated.

Total Lines of Code (LOC), or Source Lines of Code (SLOC), counts all lines, regardless of whether they contain code, comments, or are blank .
A number of 1000 is usually the maximum in a class or file.
Note that RefactorIT calculates LOC as follows: method body lines only are counted for a method; class and interface body lines only are counted for a class or interface; lines of all source files belonging to a package are counted for the package. The implications are :

- Method declarations and Javadoc comments are not counted on the method level--they are counted on the class/interface and package levels.
- Class or interface declarations and Javadoc comments are not counted on the class/interface level--they are counted on the package level.

Example

```
c /**
c  * Sends broadcast message
c  *
c  * @return <true> if message successfully sent;
c  *         <false> otherwise.
c  */
c public boolean broadcast(Message message) {
m   // Sends broadcast message
m   // to all registered clients
m   if (!isTransportAvailable()) {
m     return false; // Message not sent
m   }
m
m   getTransport().broadcast(message);
m
m   return true; // Message sent
c }
```

LOC for the class containing the code above is incremented by 17 due to this code (lines labeled **c** and **m**). LOC for broadcast method itself is 9 (lines labeled **m**).

See Also

Non-Comment Lines of Code
Comment Lines of Code
Density of Comments
Metrics

Non-Comment Lines of Code (NCLOC)

Based on http://jakarta.apach e.org/turbine/maven/metrics.html

Metrics derived from lines of code are perhaps the simplest of all the metrics to define and compute. Counting lines has a long history as a software metric dating from before the rise of structured programming, and it is still in widespread use today. The size of a method or a class affects the ease with which it can be understood, its re-usability and its maintainability. There are a variety of ways that the size can be calculated.

Non-Comment Lines of Code (NCLOC), or Non-Comment Source Lines (NCSL), or Effective Lines of Code (ELOC), counts all lines that are not regular comments or Javadoc comments. Blank lines are also not counted .

1000 lines is usually maximum counted in a class or file.

Note that RefactorIT calculates NCLOC as follows: only non-comment method body lines are counted for a method; only non-comment class or interface body lines are counted for classes and interfaces; non-comment lines of all source files belonging to a package are counted for the package. The implications are :

- Method declarations are not counted on the method level--they are counted on the class/interface and package levels.
- Class or interface declarations are not counted on the class/interface level--they are counted on the package level.

Example

```
  /**
   * Sends broadcast message
   *
   * @return <true> if message successfully sent;
   *         <false> otherwise.
   */
c public boolean broadcast(Message message) {
    // Sends broadcast message
    // to all registered clients
m   if (!isTransportAvailable()) {
m     return false; // Message not sent
```

```
m   }

m   getTransport().broadcast(message);

m   return true; // Message sent
c }
```

NCLOC for the class containing the code above is incremented by 7 due to this code (lines labeled **c** and **m**). NCLOC for the broadcast method itself is 5 (lines labeled **m**).

See Also

- Total Lines of Code
- Comment Lines of Code
- Density of Comments
- Metrics

Comment Lines of Code (CLOC)

Based on http://jakarta.apach e.org/turbine/maven/metrics.html

Metrics derived from lines of code are perhaps the simplest of all the metrics to define and compute. Counting lines has a long history as a software metric, dating from before the rise of structured programming, and it is still in widespread use today. The size of a method or a class affects the ease with which it can be understood, its re-usability, and its maintainability. There are a variety of ways that the size can be calculated.

Comment Lines of Code (CLOC) counts all lines that contain regular comments or Javadoc comments. Empty lines within both kinds of comments are also counted.

Note that RefactorIT calculates CLOC as follows: only comments inside a method body are counted for a method; only regular comments and Javadoc comments inside a class or interface body are counted for classes and interfaces; regular comments and Javadoc comments of all source files belonging to a package are counted for the package. The implications are:

- Javadoc comments for a method are not counted on the method level--they are counted on the class/interface and package levels.

- Javadoc comments for a class or interface are not counted on the class/interface level--they are counted on the package level.

Example

```
c /**
c  * Sends broadcast message
c  *
c  * @return <true> if message successfully sent;
c  *         <false> otherwise.
c  */
  public boolean broadcast(Message message) {
m   // Sends broadcast message
m   // to all registered clients
    if (!isTransportAvailable()) {
m     return false; // Message not sent
    }

    getTransport().broadcast(message);

m   return true; // Message sent
  }
```

CLOC for the class containing the code above is incremented by 10 due to this code (lines labeled **c** and **m**). CLOC for broadcast method itself is 4 (lines labeled **m**).

See Also

- Total Lines of Code
- Density of Comments
- Non-Comment Lines of Code
- Metrics

Density of Comments (DC)

Based on http://jakarta.apache.org/turbine/maven/metrics.html

Metrics derived from lines of code are perhaps the simplest of all the metrics to define and compute. Counting lines has a long history as a software metric dating from before the rise of structured programming, and it is still in widespread use today. The size of a method or a class affects the ease with which it can be understood, its re-usability, and its maintainability. There are a variety of ways that the size can be calculated.

Density of Comments (DC) provides the ratio of comment lines to all lines. Thus, DC = CLOC / LOC.

Note that RefactorIT calculates CLOC as follows: only comments inside a method body are counted for a method; only regular comments and Javadoc comments inside a class or interface body are counted for classes and interfaces; regular comments and Javadoc comments of all source files belonging to a package are counted for the package. The implications are:

- Javadoc comments for a method are not counted on the method level -- counted on the class/interface and package levels. Thus, a method with javadoc but without any comments in body will have DC of 0.
- Javadoc comments for a class or interface are not counted on the class/interface level--they are counted on the package level. Thus, a class or interface with javadoc but without any comments or javadocs in body will have DC of 0.

See Also

Total Lines of Code
Non-Comment Lines of Code
Comment Lines of Code
Metrics

Number of Parameters (NP)

This metric counts the number of parameters of a method or a constructor.

In general, methods and constructors with more than three parameters are hard to remember and to use. Source code with invocations of methods that have long parameter lists is hard to

comprehend. Additionally, invoking methods with similar signatures is error-prone.

See Also

Metrics

Executable Statements (EXEC)

Counts the number of executable statements.
Executable statement is a statement specifying an explicit action to be taken. Also known as imperative statement.

Example

```
int a = 13;
int b;
b = a + 4; // <- executable statement
a++, b++; // <- two executable statements
if (b < 6) {
  System.out.println( // <- executable statement
    "Hello, World");
}
```

See Also

Metrics

Cyclomatic Complexity (V(G))

Based on http://jakarta.apache.org/turbine/maven/metrics.html and http://www.sei.cmu.edu/str/descriptions/cyclomatic.html

Cyclomatic complexity is the most widely used member of a class of static software metrics. Cyclomatic complexity can be considered a broad measure of soundness and confidence for a program. Introduced by Thomas McCabe in 1976, cyclomatic complexity measures the number of linearly-independent paths through a program module. This measure provides a single ordinal number that can be compared to the complexity of other programs. Cyclomatic complexity is often referred to simply as program complexity, or as McCabe's complexity. It is often used in concert with other software metrics. As one of the more widely-accepted software metrics, it is intended to be independent of language and language format. The metric can additionally be interpreted as the cost of producing test cases for the code.

Methods with a high cyclomatic complexity tend to be more difficult to understand and maintain. In general the more complex the methods of an application, the more difficult it is to test it, which adversely affects the application's reliability.

V(G) is a measure of the control flow complexity of a method or constructor. It counts the number of branches in the body of the method, defined as:

- while statements
- if statements
- for statements
- conditions such as && and ||

This metric is computed as follows:

- each function has a base complexity of 1
- each atomic condition adds 1
- each case block of switch adds 1.

Example

```
... // in the beginning: V(g) = 1

// +2 conditions, V(g) = 3:
if ((i > 13) || (i < 15)) {

  System.out.println("Hello, there!");

  // +3 conditions, V(g) = 6:
  while ((i > 0) || ((i > 100) && (i < 999))) {
    ...
  }
}
```

```
// +1 condition, V(g) = 7
i = (i == 10) ? 0 : 1;

switch(a) {
  case 1: // +1, V(g) = 8
    break;
  case 2: // +1, V(g) = 9
  case 3: // +1, V(g) = 10
    break;
  default:
    throw new RuntimeException("a = " + a);
}
```

In more technical terms, the metric is the cyclomatic complexity of a flow graph of code. In this graph, nodes are branches, and edges are execution paths. V(G) is then calculated as E - N - 2, where E is the number of edges and N is the number of nodes.
A common application of cyclomatic complexity is to compare it against a set of threshold values. For example,

V(G)	Risk Evaluation
1 - 10	a simple program, without much risk
11 - 20	more complex program, moderate risk
21 - 50	complex program, high risk
greater than 50	untestable and incomprehensible program, very high risk

Another approach is to compare cyclomatic complexity of different pieces of code and thus relate their complexity.

See Also

- Weighted Methods per Class Metrics

Weighted Methods per Class (WMC)

Based on http://jakarta.apache.org/turbine/maven/metrics.html
If the number of methods in a class can be determined during the design and modeling phase of a project, this number can be used as a predictor of how much time and effort is needed to develop, debug, and maintain the project. This metric can be further refined by incorporating a weighting for the complexity of each method. The usual weighting is given by the cyclomatic complexity of the method.
RefactorIT sums the V(G) (cyclomatic complexity) of all declared methods and constructors of class to calculate WMC.

See Also

Cyclomatic Complexity
Metrics

Response for Class (RFC)

Based on http://jakarta.apache.org/turbine/maven/metrics.html
The response set of a class is the set of all methods and constructors that can be invoked as a result of a message sent to an object of the class. This set includes the methods in the class's inheritance hierarchy and methods that can be invoked on other objects.
The Response For Class metric is defined to be the size of the response set for the class. A class with a larger response set is considered to be more complex than a class with a smaller response set. For example, if a method call on a class can result in a large number of different method calls on the target and other classes, then it can be harder to test the behavior of the class and to debug problems. Testing and debugging will typically require a deeper understanding of the

potential interactions that objects of the class can have with the rest of the system.

See Also

Metrics

Depth in Tree (DIT)

Based on http://jakarta.apache.org/turbine/maven/metrics.html

This metric calculates how far down the inheritance hierarchy a class is declared. DIT is the distance from class to root of the inheritance hierarchy. In Java, all classes have java.lang.Object as their ultimate superclass, which is defined to have a depth of 0. So a class that immediately extends java.lang.Object has a metric value of 1; any of its subclasses will have a value of 2, and so on.

A class that is deep within the tree inherits more methods and state variables, thereby increasing its complexity and making it difficult to predict its behavior. It can be harder to understand a system with many inheritance layers.

DIT is defined as follows for classes and interfaces:

- all interface types have a depth of 1
- the class java.lang.Object has a depth of 0
- all other classes have a depth of 1 + the depth of their super class

See Also

Number of Children

Metrics

Number of Children (NOC)

This metric measures the number of direct subclasses of a class.
It is assumed that the more children a class has, the more responsibility there is on the maintainer of the class not to break their behavior. As a result, it is harder to modify such a class.

See Also

- Depth in Tree
- Metrics

Number of Types (NOT)

This metric reports the number of classes and interfaces.
Inner classes and interfaces, as well as local classes and interfaces, are reported.

See Also

- Number of Abstract Types
- Number of Concrete Types
- Number of Exported Types
- Metrics

Number of Abstract Types (NOTa)

This metric reports the number of abstract classes and interfaces.

See Also

- Number of Types
- Number of Concrete Types
- Number of Exported Types
- Metrics

Number of Concrete Types (NOTc)

This metric reports the number of concrete (non-abstract) classes.

See Also

- Number of Types
- Number of Abstract Types
- Number of Exported Types
- Metrics

Number of Exported Types (NOTe)

This metric reports the number of classes and interfaces that are exported outside of package.

See Also

- Number of Types
- Number of Abstract Types
- Number of Concrete Types
- Metrics

Afferent Coupling (Ca)

This metric counts number of classes and interfaces inside a package that classes and interfaces of other packages depend on.
For a technical article, see *Stability*, by Robert C. Martin (PDF) (http://www.objectmentor.com/resources/articles/stability.pdf).

See Also

- Efferent Coupling
- Abstractness
- Instability
- Distance from Main Sequence
- Metrics

Efferent Coupling (Ce)

This metric counts number of classes and interfaces in other packages that that classes and interfaces of a package depend on.
For a technical article, see *Stability*, by Robert C. Martin (PDF) (http://www.objectmentor.com/resources/articles/stability.pdf).

See Also

- Afferent Coupling
- Abstractness
- Instability
- Distance from Main Sequence
- Metrics

Abstractness (A)

This metric counts the ratio of abstract classes and interfaces for a package.
For a technical article, see *Stability*, by Robert C. Martin (PDF) (http://www.objectmentor.com/resources/articles/stability.pdf).

See Also

- Distance from Main Sequence
- Metrics

Instability (I)

This metric was proposed by Robert C. Martin and applies to packages. The metric measures degree of instability (the opposite of stability) of a package. (For the original technical article, see *Stability*, by Robert C. Martin, in PDF format at http://www.objectmentor.com/resources/articles/stability.pdf.)

A package might be changed mainly for two reasons:

- There are parts of the package that clients do not use.
- Some packages that the package depends on have changed, and those changes cannot be hidden inside this package

One extreme case is when the package is not used by any other packages, but depends itself on other packages. Changes to such a package can be easily justified, either because the changes are easy or because the changes are necessary. In both cases there is no impact on the package's clients, since there aren't any. As a result, there are no obstacles to changing the package, which is why such a package is called completely instable.

Another extreme case is when the package is used by clients, but doesn't itself depend on any other packages. There are no pressures to change such a package resulting from changes in the packages it uses, since there aren't any dependencies. Additionally, as there are many existing clients using the package, there is pressure not to change it because clients will have to be modified. That is why such a package is called completely stable.

Robert C. Martin proposed the Stable Dependencies Principle, which states that the dependencies between packages in a design should be in the direction of the stability of the packages. In other words, a package should depend only on packages that are more stable than it is.

The instability metric is calculated as: I = Ce / (Ca + Ce). It falls in the range between 0 and 1. 0 indicates a maximally stable package, and 1 indicates a maximally instable package.

See Also

- Abstractness
- Afferent Coupling
- Efferent Coupling
- Distance from Main Sequence
- Metrics

Distance from Main Sequence (D)

This metric was proposed by Robert C. Martin and applies to packages. (For a technical article, see *Stability*, by Robert C. Martin (PDF) (http://www.objectmentor.com/resources/articles/stability.pdf.) The primary rationale is that abstractness and stability of packages are closely connected. Namely, the more abstract a package is, the more stable it should be, since it should have many clients that depend on its abstractions.

Packages that are maximally stable (I = 0) and at the same time maximally abstract (A = 1) are good ones. Similarly, packages that are completely unstable (I = 1) and at the same time concrete (A = 0) are also good packages. Other packages should be located on a straight line connecting these two points on an I-A plot. This line is called the Main Sequence.

The D metric measures the square of the distance of the package from the Main Sequence. D = |A + I| - 1.

A package with A = 0, I = 0 and, thus, D = 1 is maximally stable and concrete. Such a package is not desirable, since it is rigid. It cannot be extended because it is not abstract. At the same time, it is hard to change because it is stable (thus, has clients depending on it).

A package with A = 1, I = 1 and, thus, D = 1 is maximally stable and concrete. Such a package is not desirable: It is rigid because abstractions are impossible to extend.

See Also

- Abstractness
- Instability
- Metrics

Cyclic Dependencies (CYC)

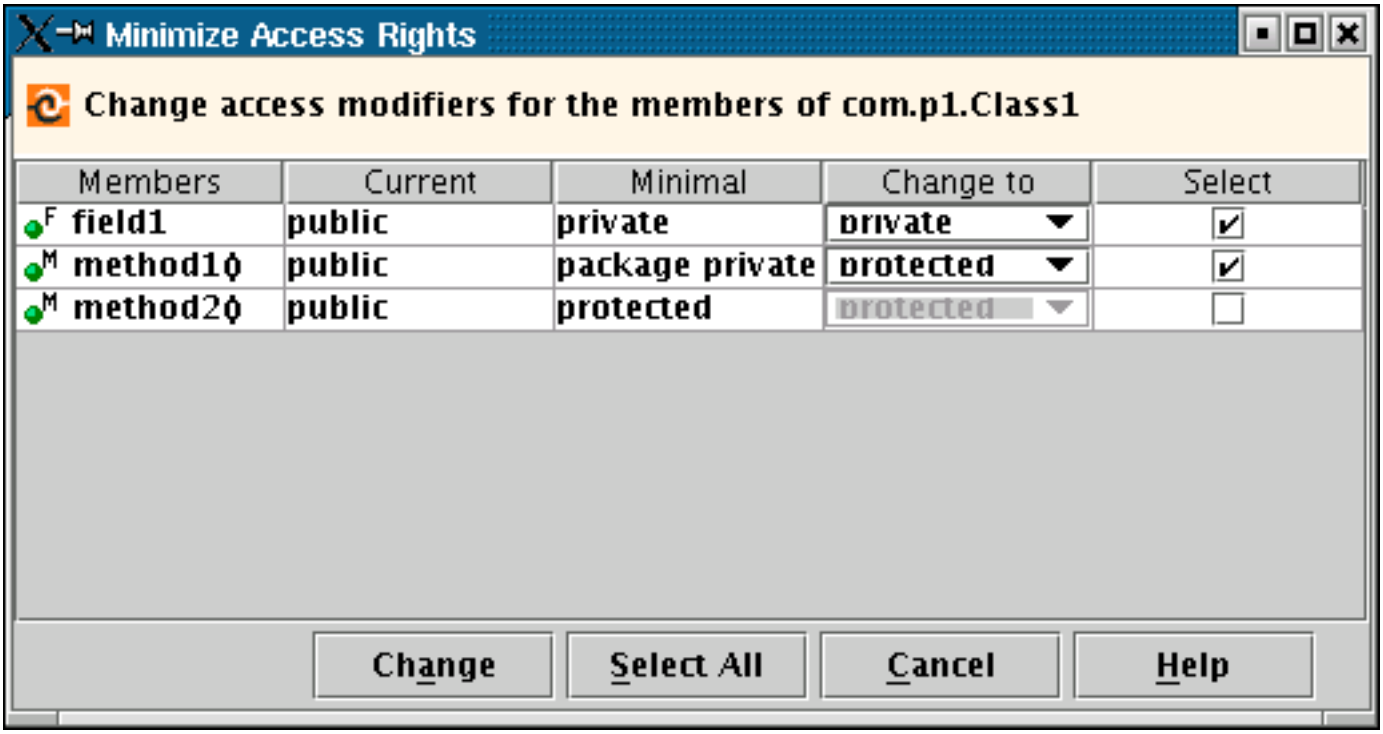
Cyclic dependencies on the method, class, and package level are maintenance nightmares and indicate potential places to apply refactoring changes.
Momentary CYC is only available on the package level.
Cyclically dependent packages are not only harder to comprehend or compile individually, but they cannot be packaged, versioned, and distributed separately. Thus, they violate the idea that a Java package is the unit of release.
Cyclic Dependencies (CYC) for a package counts a mutual dependency: the number of other packages the package depends on, and which in turn depend on the package. CYC of 0 indicates that the package is not engaged in cyclic dependencies with other packages.
When cyclic dependencies are reported for a package, the best way at the moment to determine which packages form the cycles is to use Where Used and Show Dependencies on the package. Where Used reports packages that the package is used in. Show Dependencies reports packages that the package depends on. The packages reported by both tools are engaged in cyclic dependencies with the package.

See Also

- Where Used
- Show Dependencies
- Metrics

Minimize Access Rights

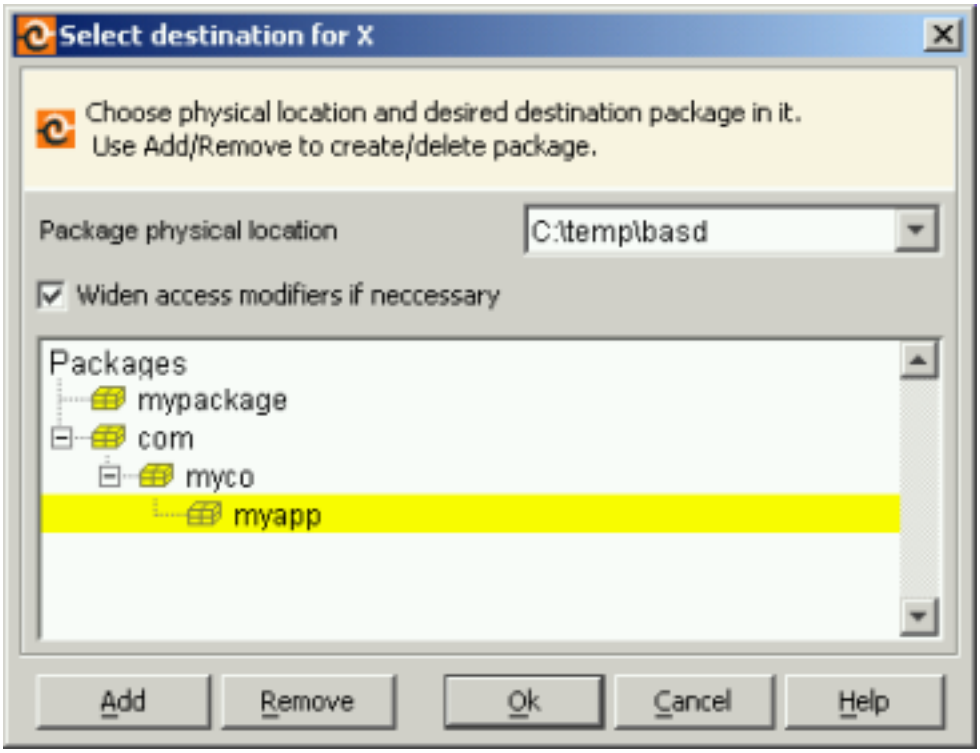
Minimizes access rights for the members of the chosen class declared in source code.



The table provides you with the name of the member, the current access, the minimal access possible, and a droplist showing possible accesse values. To change a member's access, select the checkbox in the last column and then choose a new access value from the droplist. If you want to select all members with one click, click the Select All button.

Move Class

Moves a class or interface declared in source code to another package. All references in source files are updated where necessary. If possible, the source file in which the class is declared is also relocated to follow standard package-directory mapping.
When moving a class you have to select the new package it will belong to. If no such package exists yet, you can add it in the package selection dialog. After you've selected the destination package, click OK.



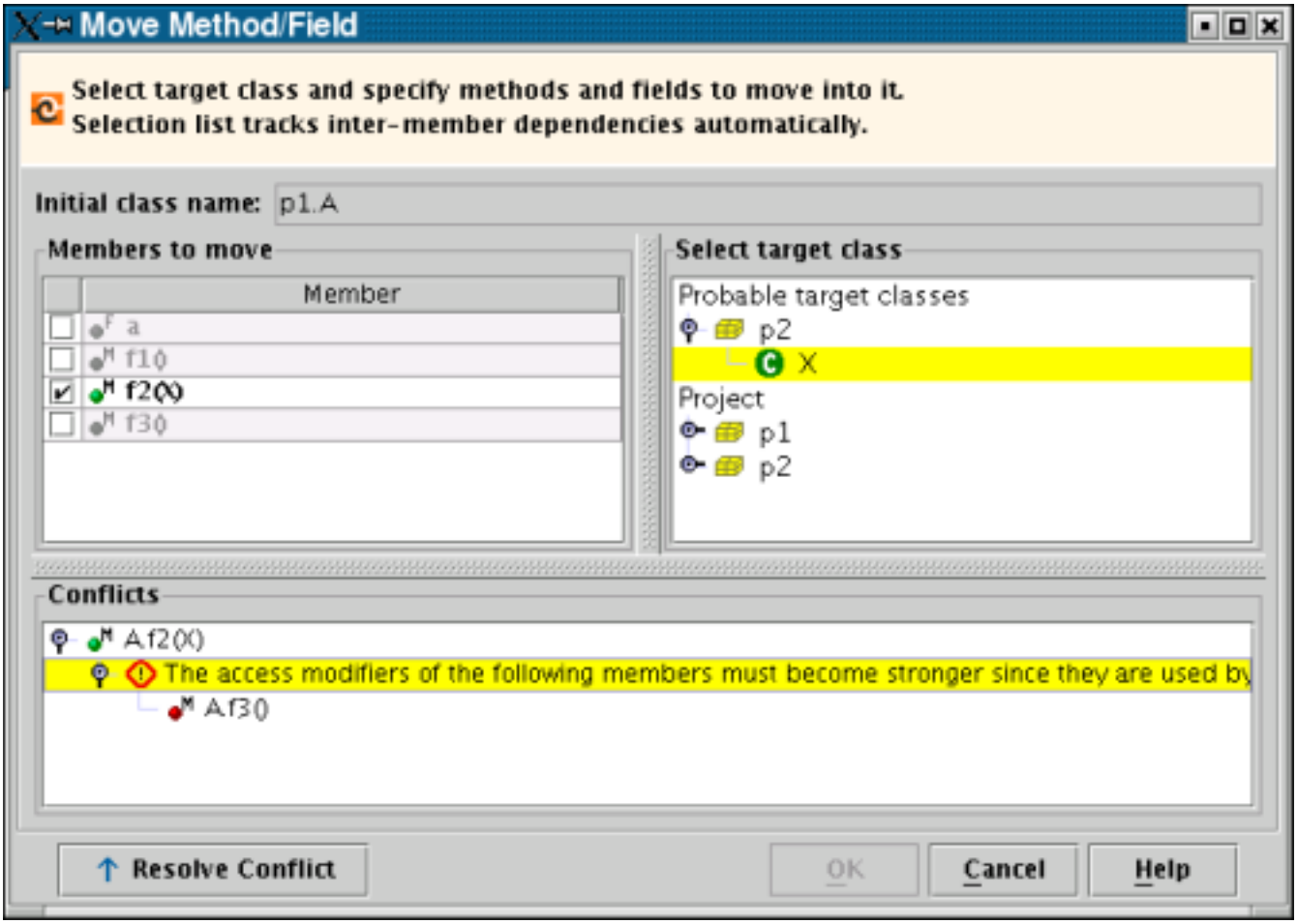
Note that moving of local classes is not supported . Neither is moving of compiled classes. But it is possible to move static inner classes (and some non-static as well), even to the same package where they already are. (In that case the class just gets extracted out of its old source file.)

See Also

Move Class in Martin Fowler's online catalog of refactoring operations

Move Method/Field

Moves a method or field declared in class to another class or interface.



Choose field or method you want to move. Choose new class or interface it will belong to. All references in classes are updated where necessary. If the move causes conflicts they will be shown in lower part of the window. They must be resolved in order to perform refactoring.

See Also

- Conflicts
- Change of the Program Functionality
- Move Method and Move Field in Martin Fowler's online catalog of refactoring operations

Conflicts

Conflict can be:
The moving of fields or methods can cause conflicts which are displayed in lower part of the window.

Resolvable

If marked by

icon that means that a few resolutions exist for this conflict. Select conflict, press 'Resolve Conflict' button and choose from the list of possible resolutions.
If marked by

icon that means that the only resolution exists for this conflict. Select conflict and press 'Resolve Conflict' button to resolve it.

Unresolvable

Is marked by

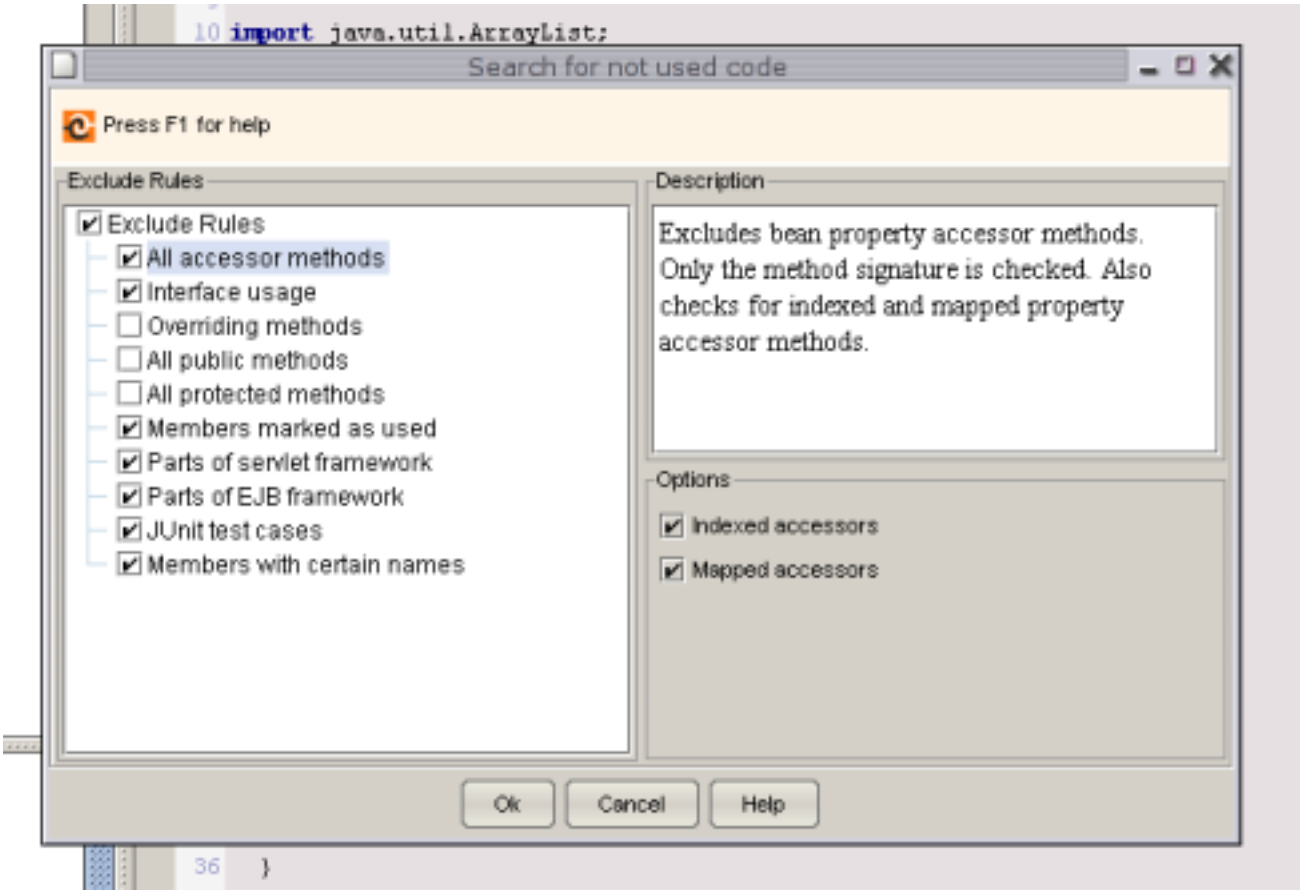
icon. These conflict cannot be resolved by RefactorIT. If such conflict emerges refactoring cannot be continued.

Not Used

Not Used finds types, members and constructors that are not used inside a project. In most cases types, members, and constructors reported by Not Used can be safely deleted from the project. Note that there are limitations that you need to be aware of.
Not Used is to some degree the reverse of Where Used. To gain a better understanding of when types, members, and constructors are reported as not in use, you need to understand when they are reported as used by Where Used.
Results are reported in two major groups: types that are not in use at all, and members and constructors that are not in use. A type is reported to be not in use only when no members of the type are used outside the type and the type is not used anywhere in code (for example, in new or throws statement). If a type is reported as not in use, you can probably delete it as a whole. If only members or constructors of a type are reported, you can delete only `.
Excluding filters

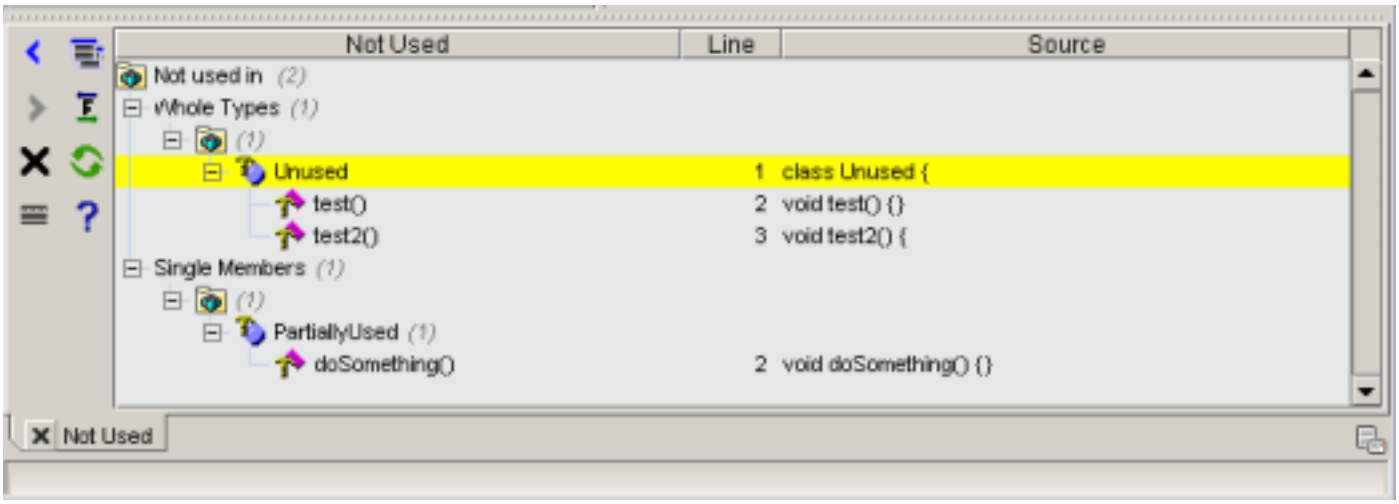
A set of filters can be used to exclude certain classes or class members from being displayed as not used. Most of the filters exclude methods or classes that are usually invoked through reflection and thus not discovered as being used. Following filters are available:

- Interface usage: Excludes methods which directly or indirectly override methods that are used.
- Overriding methods: Excludes all overriding methods, i.e. methods that are declared in a supertype.
- All public methods:Excludes all public classes and class members.
- All protected methods: Excludes all protected classes and class members.
- All accessor methods: Excludes bean property accessor methods. Only the method signature is checked. Can also check for indexed and mapped property accessor methods.
- Members marked as used: Excludes all members that have a @used tag in javadoc. This is useful if you know that a class or method is being used, but it cannot be detected by RefactorIT.
- Parts of servlet framework: Excludes all methods that override servlet methods, and all classes that implement or override servlet classes (javax.servlet.Servlet, javax.servlet.http.HttpServlet, javax.servlet.http.GenericServlet).
- Parts of EJB framework: Excludes methods that are called by Enterprise JavaBeans (EJB) containers. Also excludes classes that implement EJB interfaces (EntityBean, SessionBean, MessageDrivenBean, EnterpriseBean).
- JUnit test cases: Excludes classes that extend junit.framework.TestCase, all test* methods and setUp and tearDown methods. Also excludes public junit.framework.Suite suite() methods in all classes.
- Members with certain names: Excludes fields and methods having specified names. The list of names can be edited in the filter panel.



Example

```
class Unused {
  void test() {}
  void test2() {
    test();
    PartiallyUsed tmp = new PartiallyUsed();
    tmp.test();
  }
}

class PartiallyUsed {
  void doSomething() {}
  void test() {}
}

class Used {
  /** @USED
  */
  void markedUsed() {
  }
}
```



In this example, class Unused is not in use at all. As to class PartiallyUsed, its method doSomething is not in use either. But, as PartiallyUsed.test is in use, the whole class is not reported as not being in use; only method doSomething is.

The method markedUsed of class Used is reported as used because it's has an @USED tag in its Javadoc. As a result, also the whole class is not reported as unused.

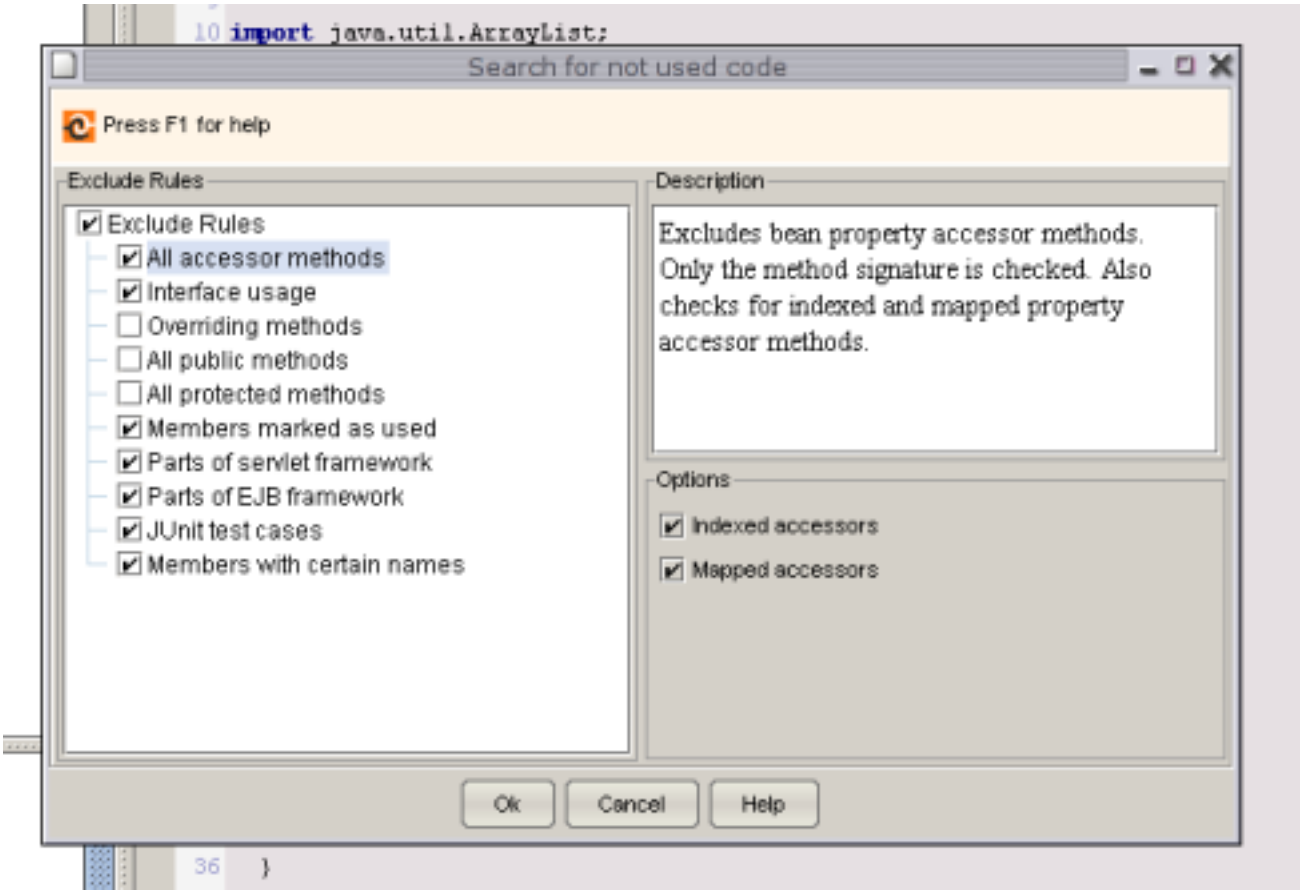
See Also

Limitations

Filter Dialog for Not Used

You can narrow down the results of a Not Used query by choosing different settings from the query's filter dialog. The filter dialog comes up automatically every time you run a Not Used query. To refine the search, you can also open that dialog later from the results panel by clicking the filter button (

) after obtaining initial results.



The following table describes the fields in this dialog box:

Name	Description
Interface usages vs. implementation usages	Interface usage means using any method in the subclasses/superclasses hierarchy that has exactly the same signature as the target method (for instance, using overriding methods counts as interface usage). Implementation usage means using a particular method body. So, overrides don't count, but simply using the method through a subclass does. (Note that this would also count as interface usage.)
Exclude public and protected members	If checked, RefactorIT will not check public or protected members.

Limitations

There are some situations when Not Used either does not report methods as not in use, or, alternatively, reports that members or types are not in use when they really are in use.
Note that, as a rule, you should think twice before removing types, methods, or constructors reported by Not Used as not in use. Due to limitations of the Java language, one cannot report usages (or lack of them) with 100% accuracy.
The following situations can cause Not Used to funciton incorrectly.

Cyclic Dependencies

```
class Test {
  private void a() {
    b();
  }

  private void b() {
    a();
  }
}

class Client {
  void test() {
    Test test = new Test();
  }
}
```

Not Used will not be able to report that both Test.a and Test.b are not in use. Test.a is only used in Test.b. Test.b is only used in Test.a. So, one could remove *both* methods simultaneously. Not Used does not report these two methods as not in use, since removing only one of them will break the code.

Limitations of Finding Usages

Not Used also has all the limitations Where Used has. For example, if a method is only invoked through the Reflection API, Not Used will report it as not in use.

See Also

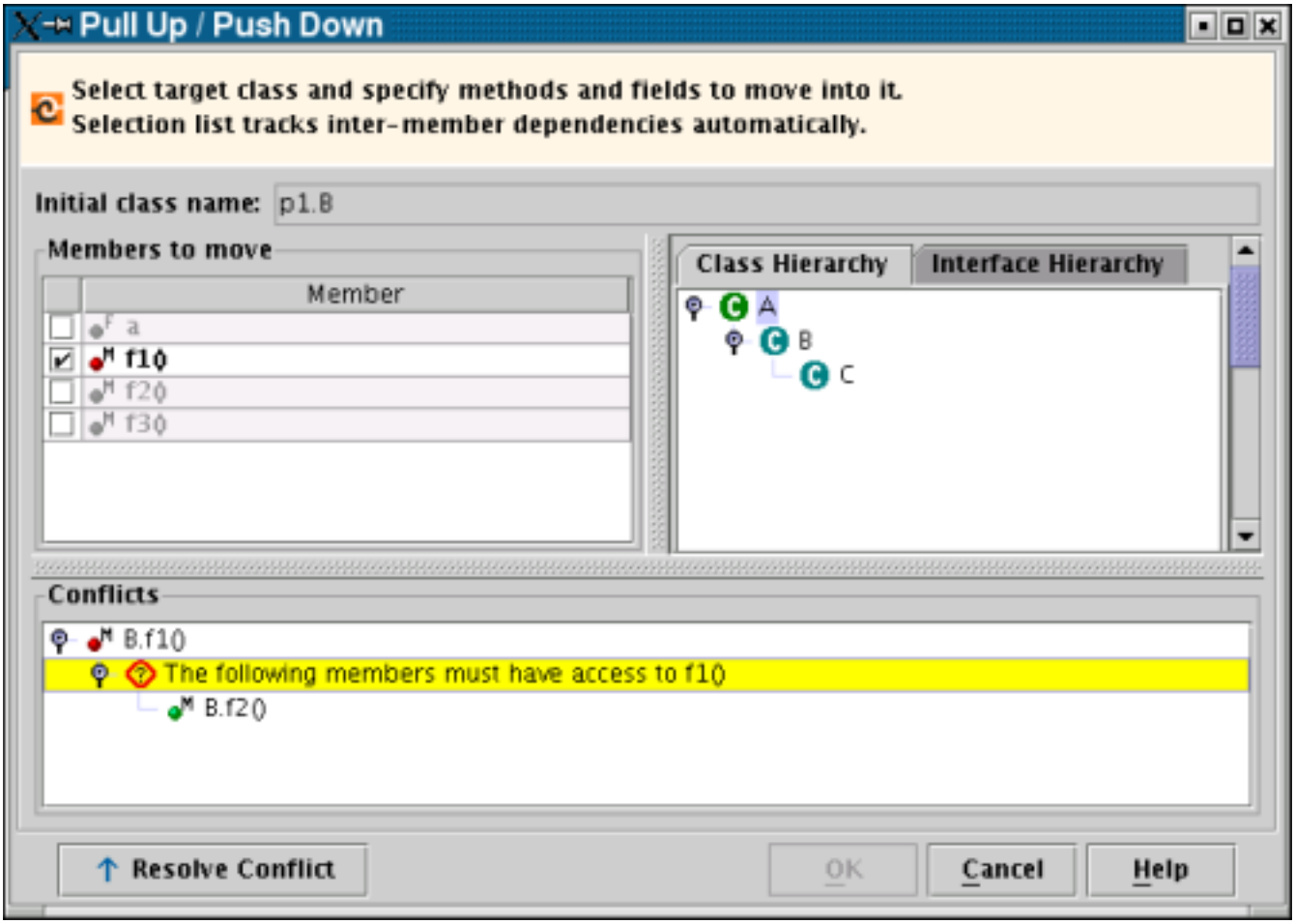
Limitations of Where Used

Override/Implement Methods

Use this refactoring to override superclass methods. Click on target class and select methods you would like to override.

Pull Up / Push Down

Pull Up / Push Down moves fields and methods to superclasses or subclasses respectively. Pulling up might be required if the field or method should be available to all subclasses, and not only to the one where it resides before refactoring. Pushing down might be required if the field or method should be available only in some, as opposed to all, subclasses of the class where it resides before refactoring.
You can run Pull Up / Push Down either on single field or method or on a class.



Choose field or method you want to move. Choose new class or interface it will belong to (tab 'Interface Hierarchy' contains tree of interfaces which currently selected class implements). All references in classes are updated where necessary. If the move causes conflicts they will be shown in lower part of the window. They must be resolved in order to perform refactoring.

See Also

Conflicts
From Martin Fowler's online catalog of refactoring operations:
Pull Up Method
Pull Up Field
Push Down Method
Push Down Field

Conflicts

Conflict can be:
The moving of fields or methods can cause conflicts which are displayed in lower part of the window.

resolvable

If marked by

icon that means that the only resolution exists for this conflict. Select conflict and press 'Resolve Conflict' button to resolve it.
If marked by

icon that means that a few resolutions exist for this conflict. Select conflict, press 'Resolve Conflict' button and choose from the list of possible resolutions.

unresolvable

Rename

Rename is one of the commonly used refactoring operations. It enables you to rename methods, fields (and all variables in general), types, and packages. All usages of the item renamed are updated correspondingly.
If the code has been compiled before renaming, it will also compile after renaming. If renaming could break your project, RefactorIT will report potential problems and ask for confirmation.
Note that there are cases where RefactorIT cannot detect that renaming will damage or even break your project. Examples include:

- The project uses reflection (for example, public static void main(String[]), servlets, EJBs, JUnit tests)
- If Javadoc comments are not updated during the rename (which is configurable from the Rename dialog box), the code resulting from the rename operation might cause warnings during Javadoc generation. Additionally, if you use tools that generate parts of code based on Javadoc comments (DbC, EJB generation), the result is undefined.

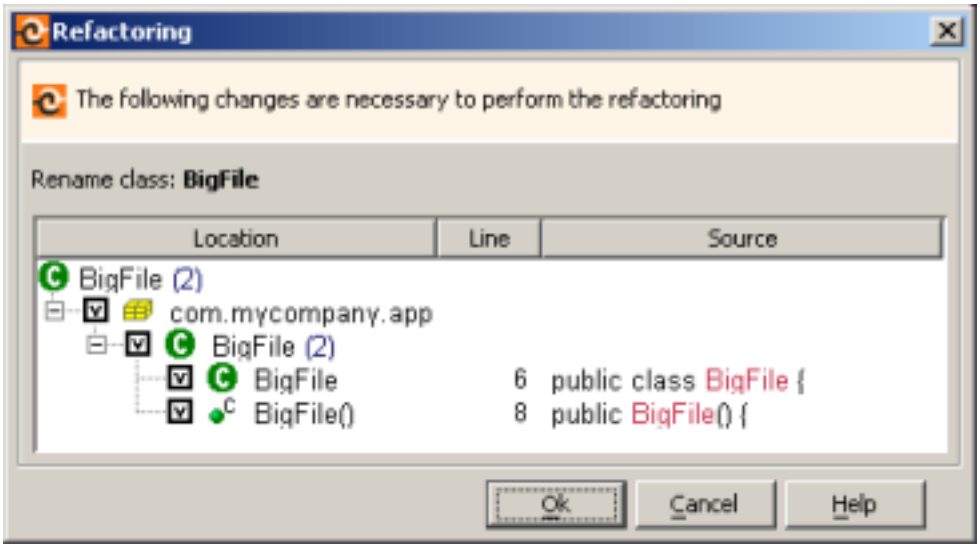
RefactorIT does its best to preserve the layout of your code. Still, there are cases where your coding conventions might be violated as a result of renaming. For example, RefactorIT does not follow the maximum source line length constraint.

See Also

Assessing Impact
Renaming a Package
Renaming a Class or InterfaceRenaming a Field
Renaming a Parameter or Local Variable
Renaming a Method
Renaming a Type

Assessing Impact

Before performing a Rename, RefactorIT lists all places in the source path that need to be changed. You can review the impact and also select which occurrences should be renamed. By default all occurrences are renamed.



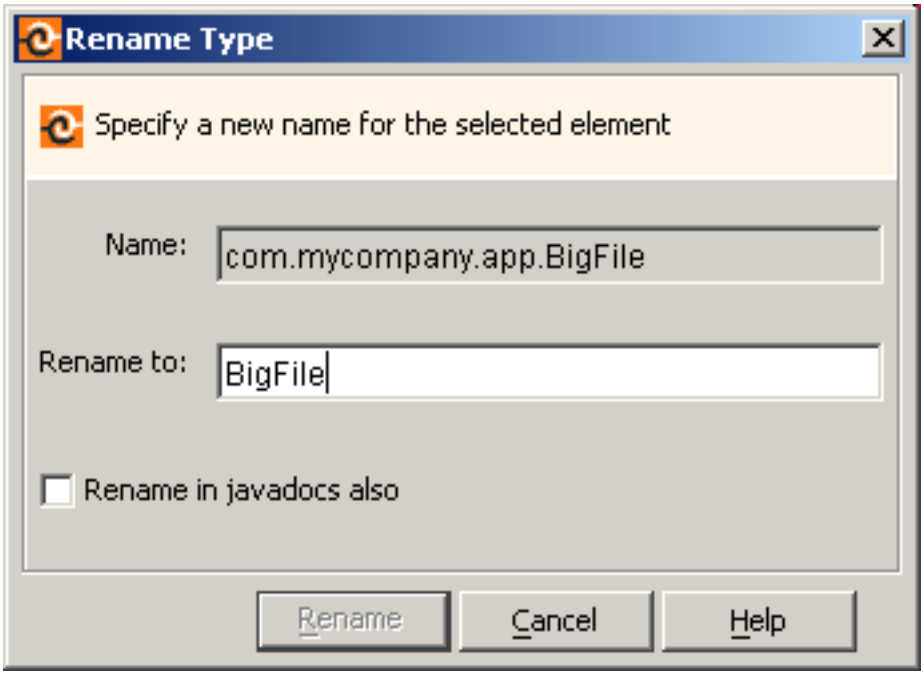
See Also

- Rename
- Renaming a Package
- Renaming a Class or Interface
- Renaming a Field
- Renaming a Parameter or Local Variable
- Renaming a Method
- Renaming a Type

Renaming a Class or Interface

Renaming a class or interface changes name of the type. When it renames a class, RefactoriIT updates constructors and all references in all source files correspondingly. You can rename a type

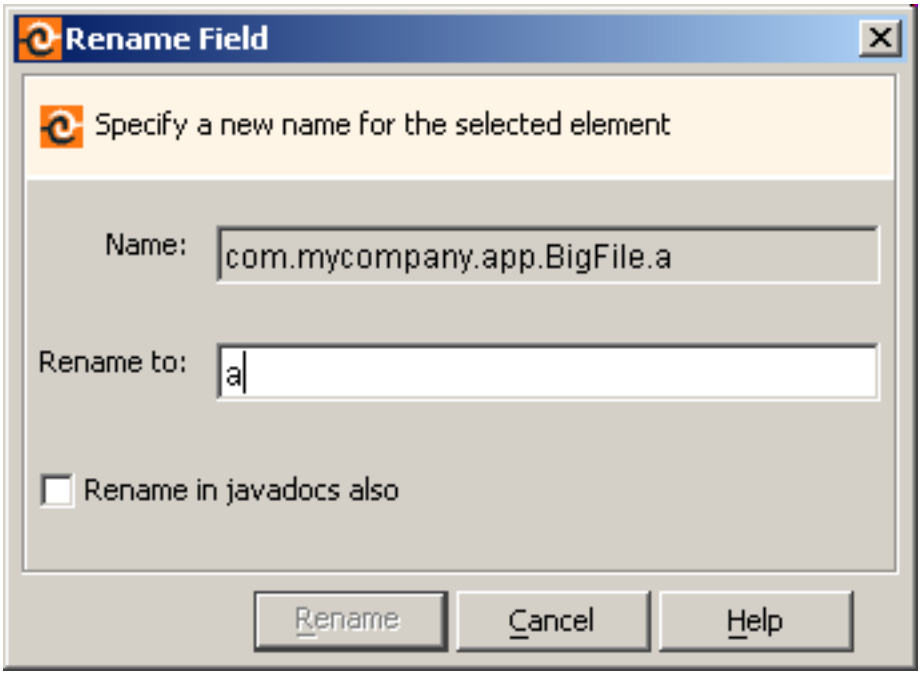
only if it is declared in source files.



See Also

- Rename
- Assessing Impact
- Renaming a Package
- Renaming a Field
- Renaming a Parameter or Local Variable
- Renaming a Method

Renaming a Field



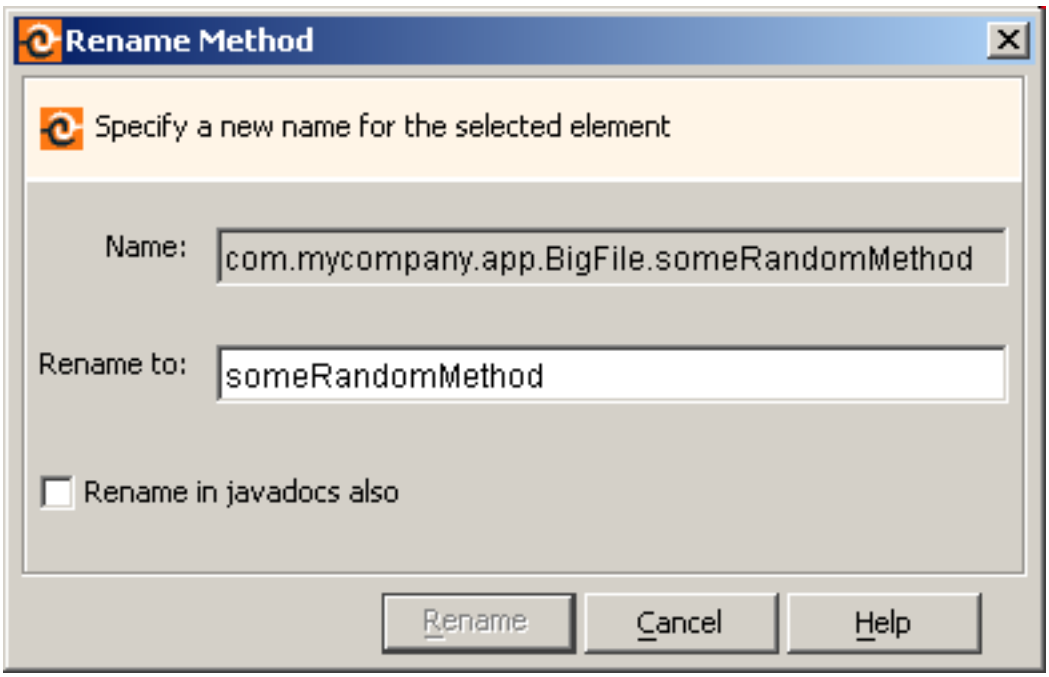
Renaming a field changes its name. All references in all source files are updated correspondingly. You can only rename a field that is declared in source files.

See Also

- Rename
- Assessing Impact
- Renaming a Package
- Renaming a Class or Interface
- Renaming a Parameter or Local Variable
- Renaming a Method
- Renaming a Type

Renaming a Method

Renaming a method changes its name. All references in source files are updated correspondingly. You can rename a method only if it is declared in source files.



There are several tricky cases to keep in mind when renaming a method. RefactorIT automatically detects the following ones:

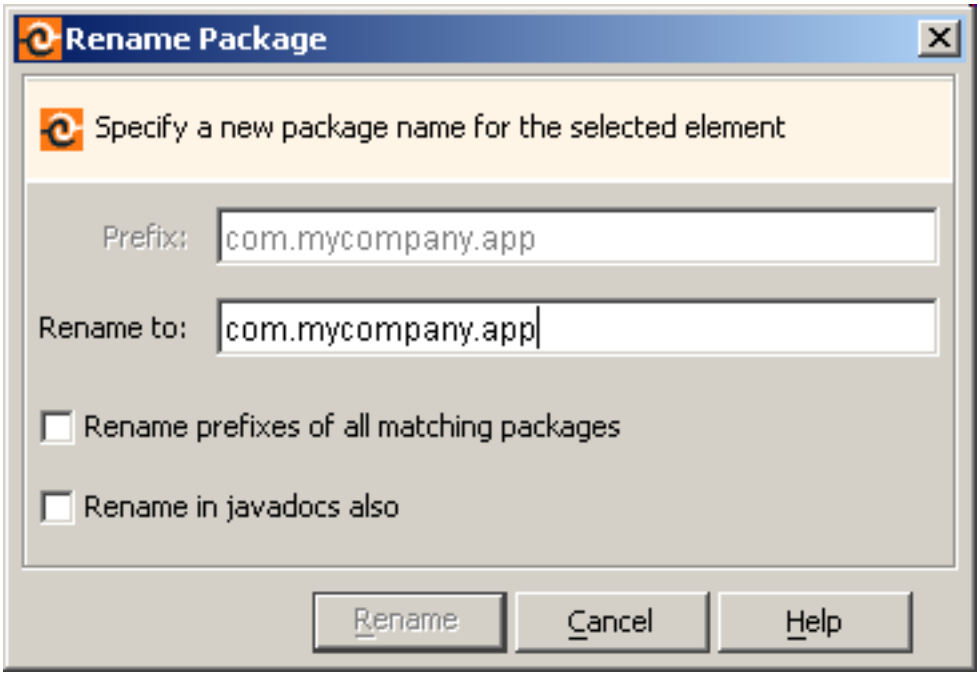
- When renaming an abstract method, RefactorIT asks if you want to rename any implementing or overriding methods.
- When renaming a method, RefactorIT asks if you want to rename any overriding methods.
- When renaming a method that overrides a method from a compiled class, RefactorIT will warn you that it is not possible to rename the overriden method because doing so might break your project.

See Also

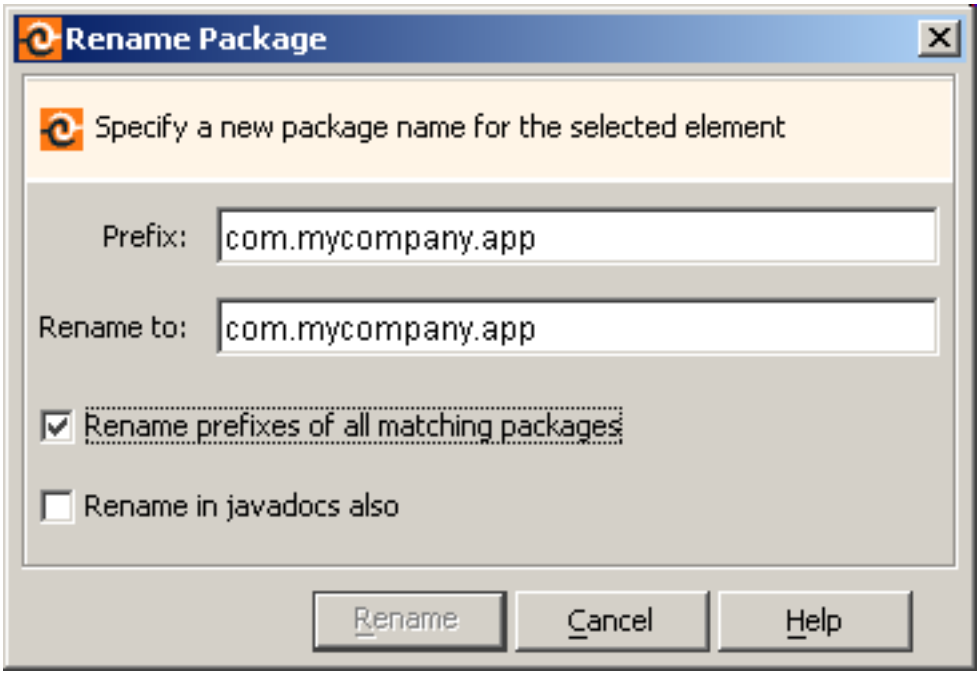
- Rename
- Assessing Impact
- Renaming a Package
- Renaming a Class or Interface
- Renaming a Field
- Renaming a Parameter or Local Variable
- Renaming a Type

Renaming a Package

Renaming a package changes the name of a package. All references (FQNs, package and import statements) in all source files are updated correspondingly.
In the following dialog box, package com will be renamed to org:



Note that with these settings package com.company will *not* be renamed because "rename prefixes of all matching packages" is turned off. (For safety reasons, this is the default behavior.) So, although packages are not actually hierarchical in Java, there is an option to treat them as such and rename all packages starting with a particular prefix, as illustrated in the following screenshot:



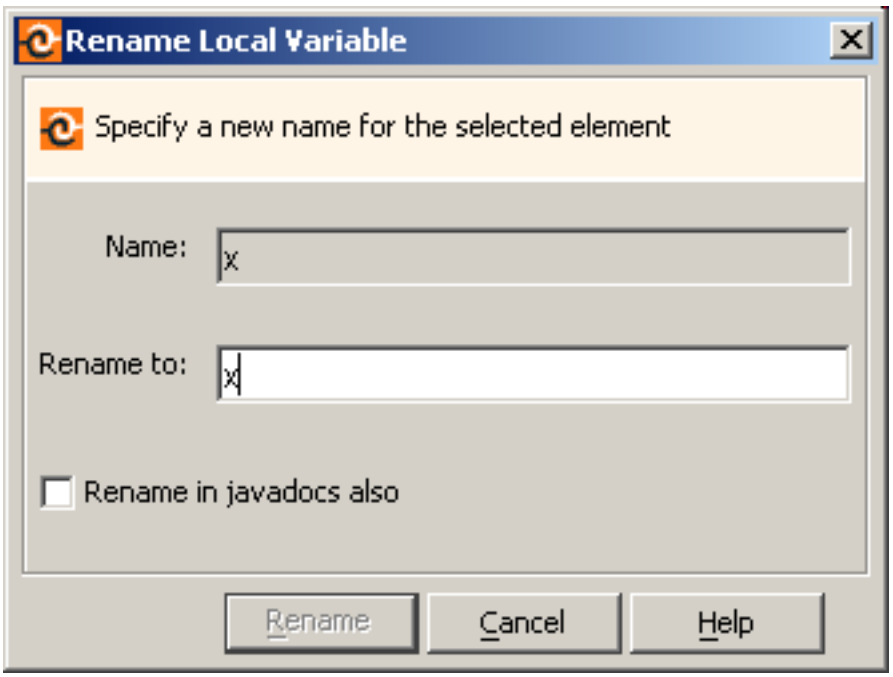
With the settings from the dialog box above, all packages that have names starting with com will be renamed so they start with org instead. The package com will be renamed to org and the package com.company will be renamed to org.company.

See Also

- Rename
- Assessing Impact
- Renaming a Class or Interface
- Renaming a Field
- Renaming a Parameter or Local Variable
- Renaming a Method
- Renaming a Type

Renaming a Parameter or Local Variable

Renaming a method parameter or a local variable changes its name. All references in the body of the method are updated correspondingly.

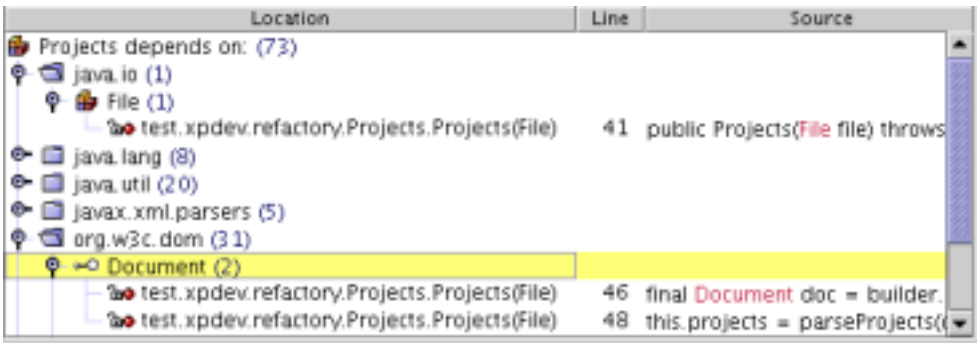


See Also

- Rename
- Assessing Impact
- Renaming a Package
- Renaming a Class or Interface
- Renaming a Field
- Renaming a Method
- Renaming a Type

Show Dependencies

Finds types, members and constructors that a type, method, or package depends on.
Show Dependencies is the reverse of Where Used. To better understand what *usages* are, check Where Used to see what it treats as usages.
Items depended upon are grouped by package, next by type depended upon, and, finally, by type, member, or constructor that depends upon the item, as shown in the following graphic:



See Also

- Limitations
- Where Used

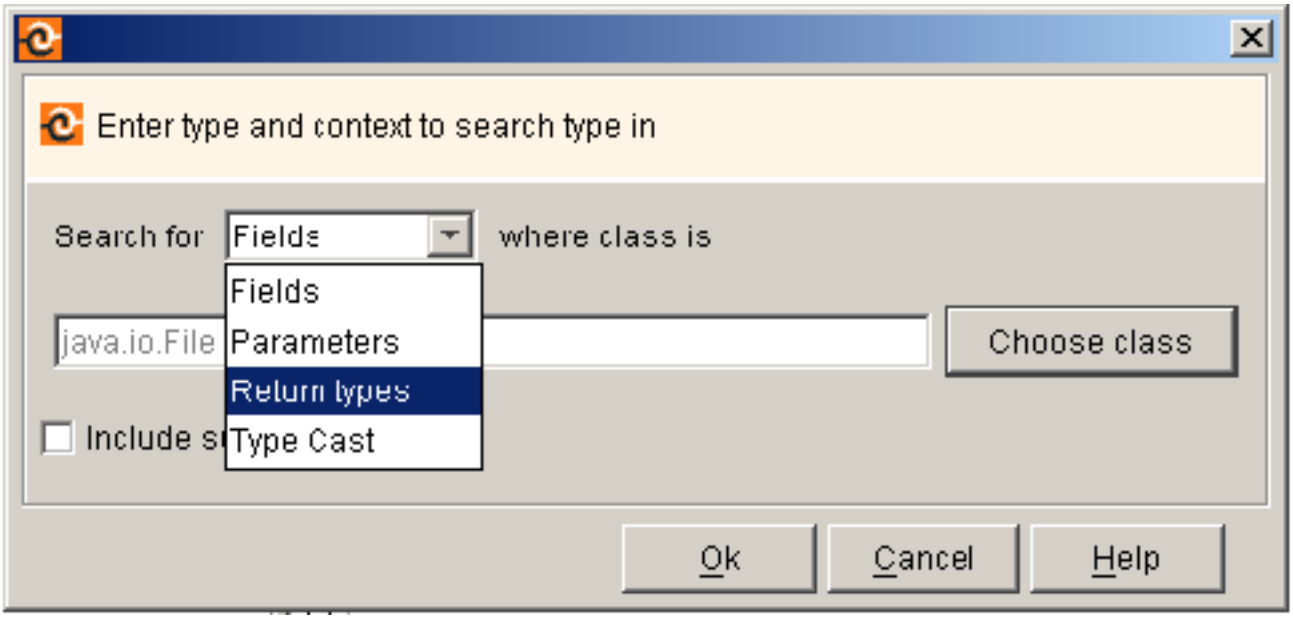
Limitations

There are some situations where Show Dependencies does not report all dependencies. These situations are caused primarily by the limitations that Java imposes on finding usages in a Java program. See Limitations of Where Used to gain more insight into the problem.

Structure Search

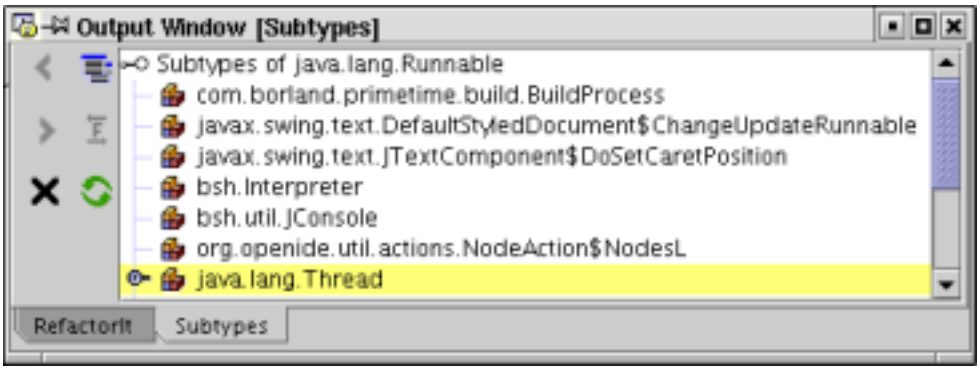
This dialog box searches for classes in certain structures.
It can search for:

- Fields of certain types
- Method parameters of certain types
- Method return types of certain types
- Type casts into certain types



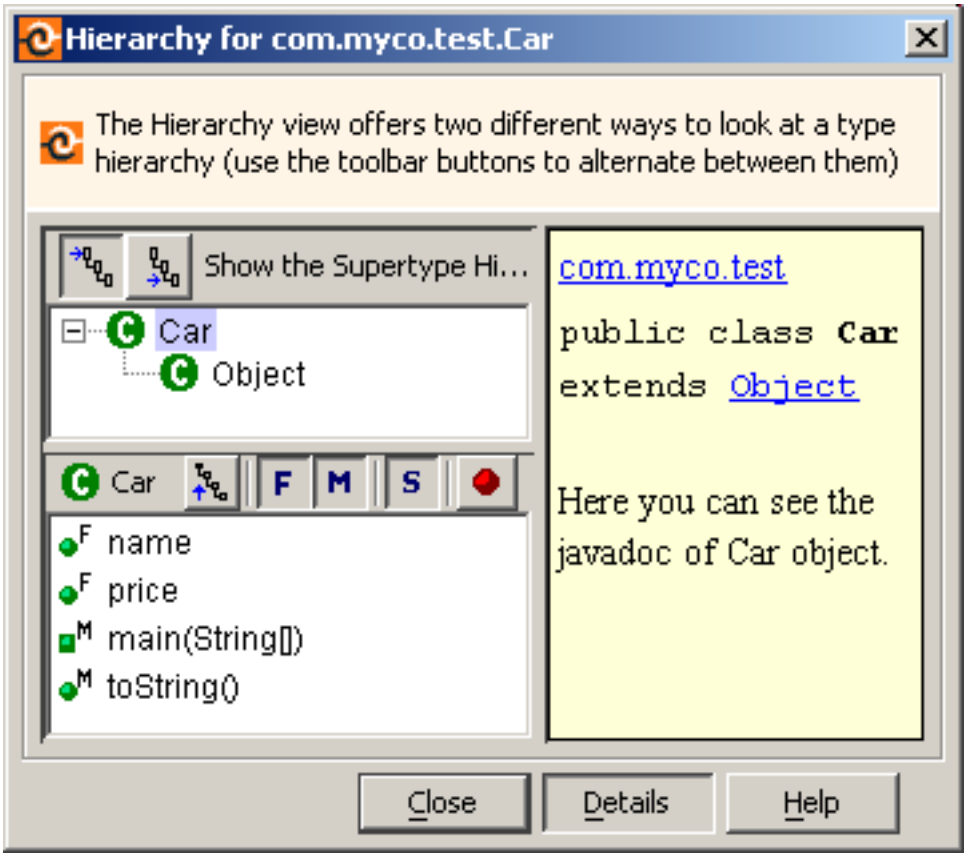
Subtypes

Subtypes shows all subtypes of a class and all implementors of an interface.



Type Info

Type Info enables you to browse classes or interface members and subtypes. Inheritance and implementation hierarchy is represented in a tree.



See Also

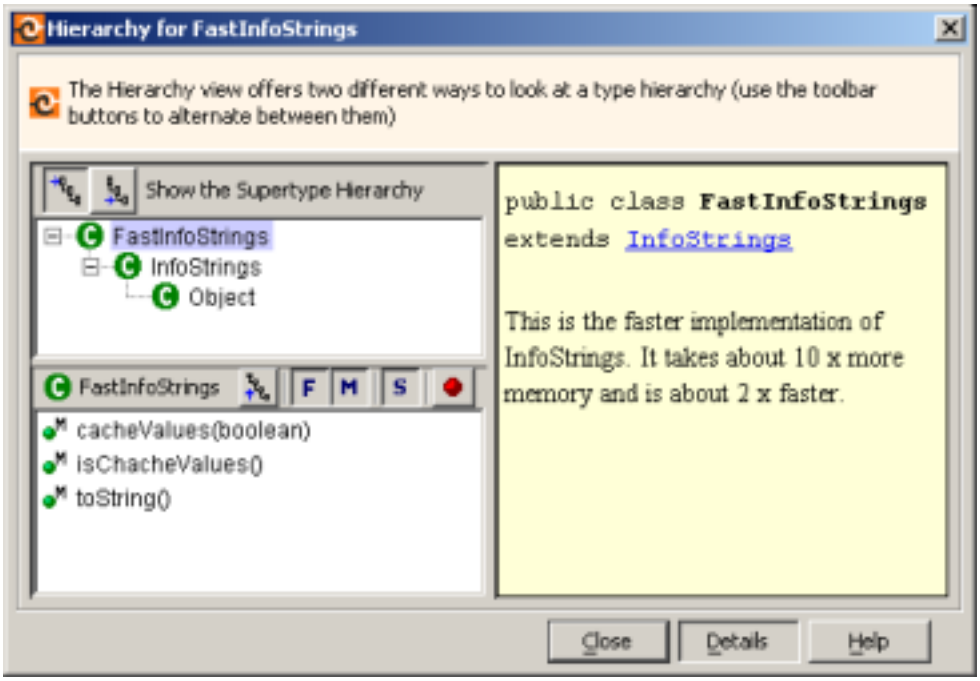
Hierarchy Mode

Hierarchy Mode

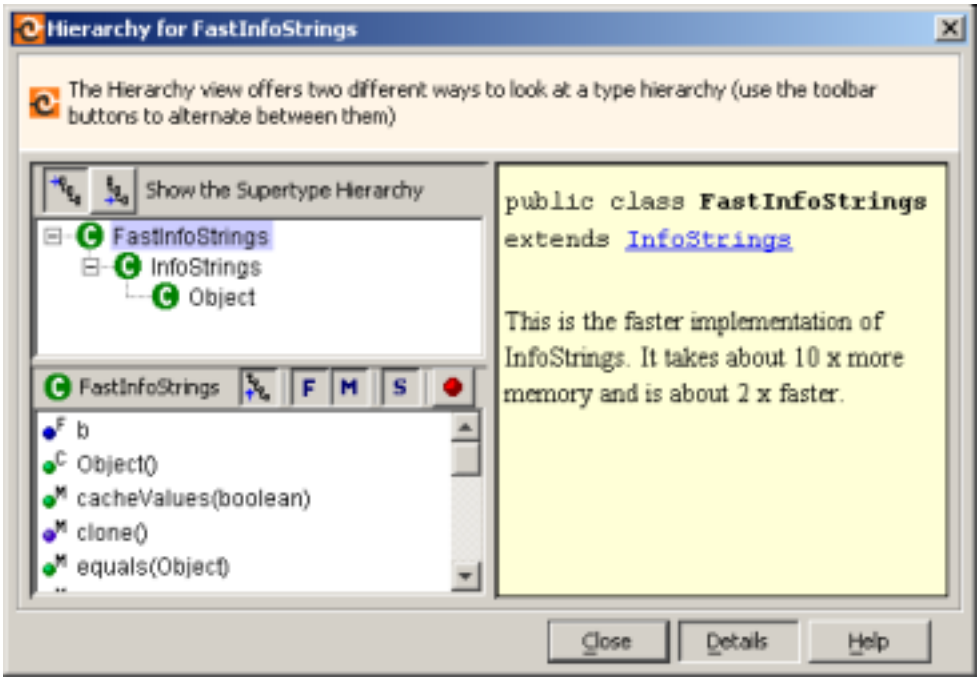
For any node in a tree shown in Type Info browser, you can choose to view only the members and constructors it declares or all accessible members and constructors. This mode is toggled by the Inherited Members button (

):

The following screenshot shows hierarchy mode set to declared members and constructors only:



The following screenshot shows hierarchy mode set to all members:



See Also

Type Info

Use Supertype Where Possible

When clicked on variable or parameter name RefactorIT will search only for possible change of variable type. When clicked on function declaration RefactorIT will search for possible return type replacement with supertype.

Use Supertype Where Possible refactoring will look for possible places to use superclass or super interface instead of target subclass. To run refactoring user must click at target class, at target variable or function and select superclass from dialog. When clicked on class or interface name RefactorIT will search for all target type usages and possible replacements in project(can be slow!).

For example we have code

```
interface MyInterface {
  void m();
}

class MyClass implements MyInterface {
  public void m() {

  }
}

class Helper {
  MyClass var;
  Helper(MyClass var) {
   this.var=(MyClass)var;
   this.var.m();
  }
  public MyClass getVar() {
    return var;

  }

  static void test() {
   MyClass a=new MyClass();
```

```
   Helper b=new Helper(a);

  }

}
```

When we apply refactoring to MyClass and choose MyInterface as supertype then after confirming changes code will look like(changes in bold):

```
interface MyInterface {
  void m();
}

class MyClass implements MyInterface {
  public void m() {

  }
}

class Helper {
  MyInterface var;
  Helper(MyInterface var) {
   this.var=(MyInterface)var;
   this.var.m();
  }
  public MyInterface getVar() {
    return var;

  }

  static void test() {
   MyInterface a=new MyClass();
   Helper b=new Helper(a);

  }

}
```

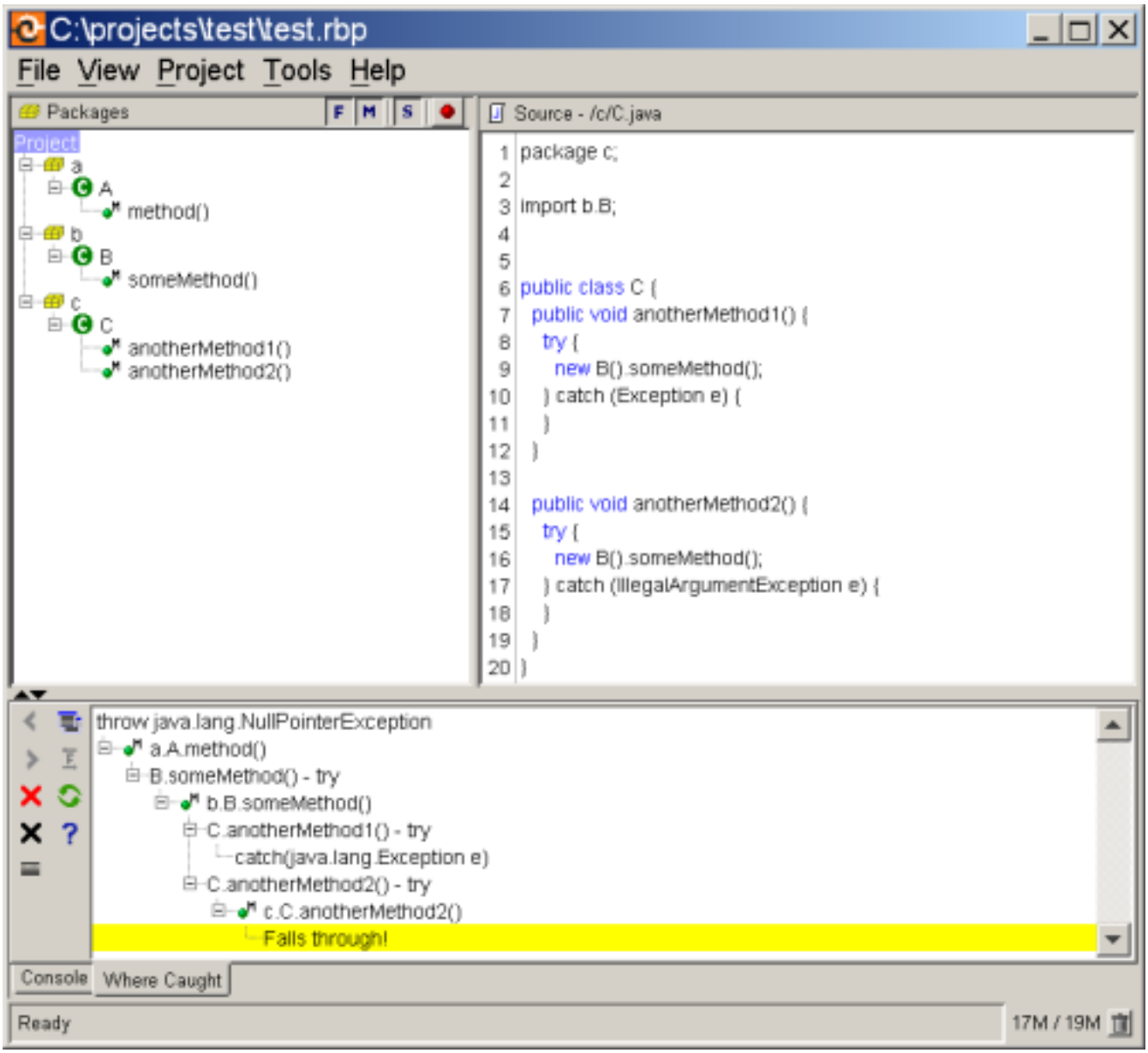
Where Caught

Finds where an exception is handled.

Can be invoked when clicked:

- on any exception name declared in the throws clause of the method or constructor declaration, e.g. when clicked on MyException string in
  public void method() throws MyException, SomeOtherException {
  ..
  }
- on the throw keyword within the body of the method or constructor, e.g.
  throw new MyException(); // click within 'throw'

Result is a tree of all the method/constructor invocation paths an exception can fly through until caught.

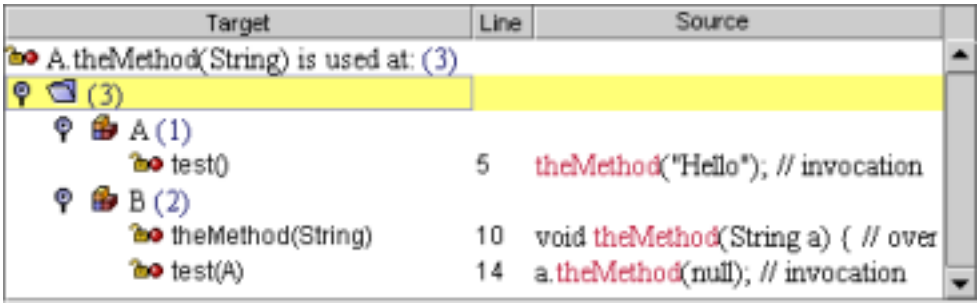


See Also

Call Tree to get better understanding of an idea of invocation paths

Where Used

Where Used finds usages of packages, types, members, and constructors inside the source path of a project. Usages in compiled classes (those in the class path) are not reported. Still, you can search for usages of compiled classes or their members.
Where Used is the reverse of Show Dependencies.
Usages are grouped by the location where they occur: first by package, next by type or source, and, finally, by line number. Additionally, places that cause usage to be reported are highlighted, as shown in the following graphic:



Double clicking a reported usage brings you to the exact location where the usage occurs. Additionally, you can right-click a package, type, method, or constructor reported in the Target column and run applicable refactoring operations on it.

See Also

Limitations

Usages of Fields
Usages of Parameters And Local Variables
Usages of Methods
Usages of Types
Usages of Packages
Filters
Show Dependencies

Limitations

There are some usages that Where Used cannot detect. You should be aware of them so you won't be surprised when they aren't detected.
Note that, as a rule, you should think twice before removing types, methods, or constructors reported by Where Used as not being in use. Due to limitations of the Java language, RefactorIT cannot report all usages (or lack of them) with 100% accuracy.

Reflection API

Invocations or instantiations via Java Reflection API are not detected. The simplest example is the public static void main(String[] params) method that is invoked via reflection. More sophisticated examples include methods imposed by the Servlet API, EJB bean implementation methods, and so on. Usually, Where Used will report no usages for such methods.

Frameworks And Libraries

If the project is a framework or library, there are probably some methods and types exposed that are not used inside the project, such as utility classes. If these classes are not used from within the project no usages are reported by Where Used.

Usages in Compiled Methods

If instances of types are passed as parameters to a compiled method, RefactorIT does not know whether some methods or fields of those instances are used in the method.
For example, a project contains several helper classes that implement java.lang.Runnable. The intent is that these types are instantiated and passed to, for example, a Thread that runs them. As

there are no invocations of the run method in the source code inside the project, no usages will be reported.
If an instance of a class is passed to a method from a compiled class (not in the source path of the project), RefactorIT does not know what happens with that object inside the method. Hence, RefactorIT does not report usages of the class's fields or methods. For example,

```
class Foo {
  public String toString() { return null; }
}

class Client {
  void test() {
    Foo foo = new Foo();
    System.out.println(foo); // foo.toString()
                             // invoked
  }
}
```

Although according to the contract of java.io.PrintStream.println(Object), foo.toString() will be invoked, no usages of foo.toString() will be reported since RefactorIT does not know what happens with instance foo inside the println method.

Subtypes

If a project contains implementations of external interfaces or subclasses of external classes, their usages might sometimes not be found. For example, a jUnit test class extends junit.framework.TestCase. Its intent is to be instantiated from outside the project and to be passed to a jUnit test runner. As RefactorIT does not know about this, it does not report usages of the class.

Where Used For Fields

Usages of a field are:

- Setting the value of the field (except during declaration)
- Getting the value of the field

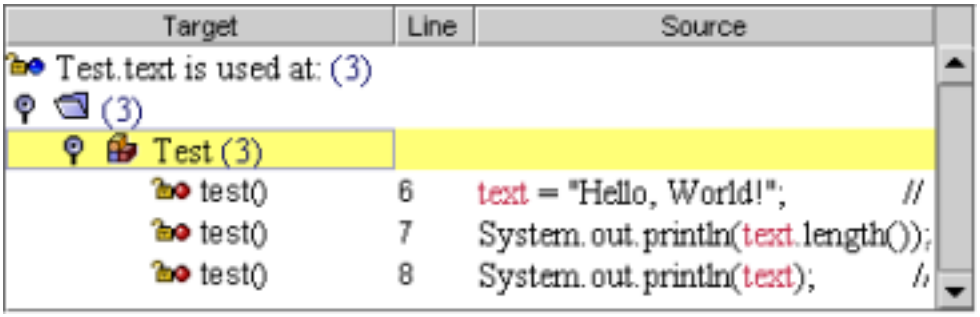
You can tune the results of the query in the filter dialog for variables that is opened automatically when you run the query.

Example

```
class Test {
  String text = "Hi!"; // Not counted as usage
                       // (declaration)
```

```
  void test() {
    text = "Hello, World!";            // set
    System.out.println(text.length()); // get
    System.out.println(text);          // get
  }
}
```

The following graphic shows three usages of Test.text:



Where Used For Methods

Usages of a method are:

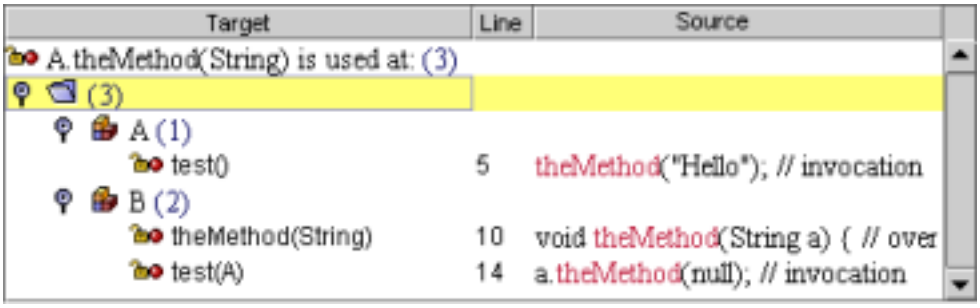
- Invocation of the method
- Declaration of overriding/implementing method in subclass

You can tune the results of the query in a filter dialog for methods. (One such dialog is for private methods and constructors, and another one is for non-private methods). The dialog will be opened automatically when you run the search.

Example

```
class A {
  void theMethod(String tmp) {}

  void test() {
    theMethod("Hello"); // invocation
  }
}

class B extends A {
  void theMethod(String a) { // overriding
  }                          // A.theMethod

  void test(A a) {
    a.theMethod(null); // invocation
  }
}
```

The following graphic shows three usages of A.theMethod(String):



Where Used For Packages

Usages of a package are usages of types declared in the package.
You can tune the results of the query in the filter dialog for packages that is opened automatically when you run the query.

Where Used For Parameters and Local Variables

Similar to usages of fields, usages of parameters and local variables are:

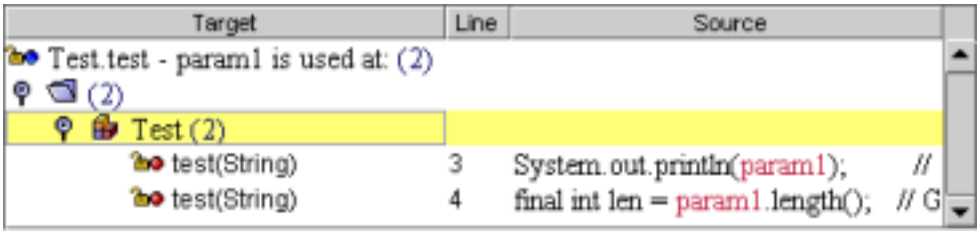
- Setting the value of the parameter or variable (except during declaration)
- Getting the value of the parameter or variable

You can tune the results of the query in the filter dialog for variables that is opened automatically when you run the query.

Example

```
void test(String param1) {
  System.out.println(param1);        // get
  final int len = param1.length();   // get
       // Setting value of "len" not counted
       // as usage of "len" because the
       // value is set in declaration.
  System.out.println(len);           // get
}
```

As shown in the following graphic, this code has:
2 usages of param1.
1 usage of len.



Where Used For Types

Usages of a type (class/interface) are:

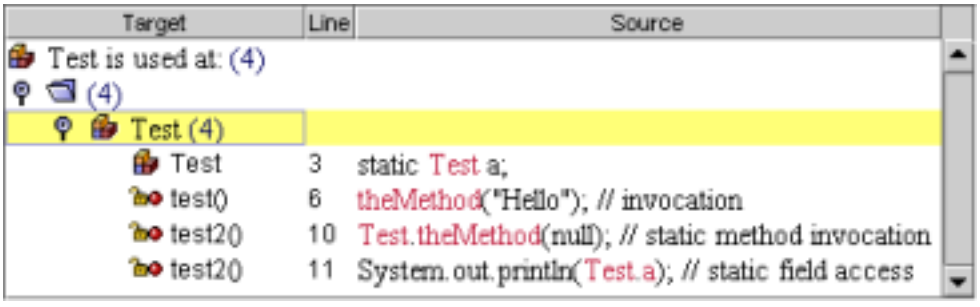
- Usages of its members
- Usages of type when performing static method invocation or static field access
- Usage of type when declaring field, parameter, local variable type
- Usage of type in import, new statements
- Usage of type in extends, implements, throws clauses
- Usage of type in type cast expressions
- etc...

You can tune the results of the query in the class filter dialog (or in the interface filter dialog). The dialog is opened automatically when you run the query.

Example

```
class Test {
  static void theMethod(String tmp) {}
  static Test a;

  void test() {
    theMethod("Hello"); // usage
  }

  void test2() {
    Test.theMethod(null); // usage
    System.out.println(Test.a); // usage
  }
}
```

The following graphic indicates that there were four usages of class Test:



Filters

You can narrow down the results of a Where Used query by choosing different settings from the query's filter dialog. The filter dialog comes up automatically every time you run a Where Used query.

You can also open that dialog later from the results panel to refine the search by clicking on the filter button

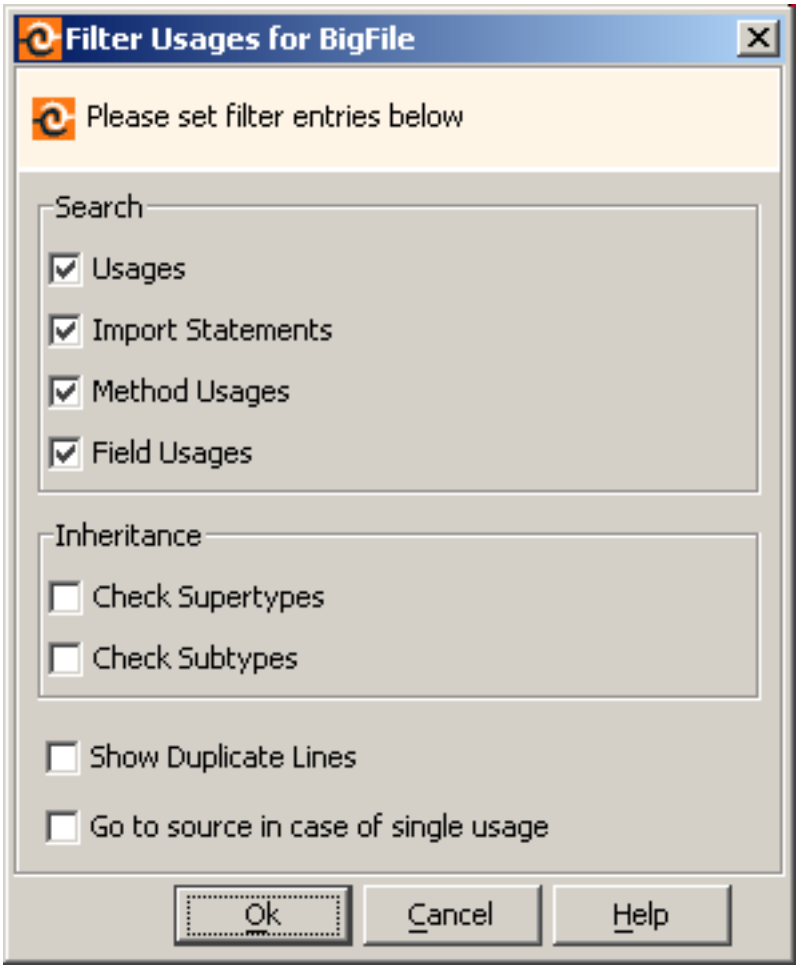
after you have obtained initial results.
RefactorIT has different specialized filters for different queries:

- Filter for class usages
- Filter for interface usages
- Filter for usages of non-private methods
- Filter for packages usages
- Filter for variable usages
- Filter for constructor and private method usages

Filter Dialog for Classes

You can narrow down the results of a Where Used query by choosing different settings from the query's filter dialog. The filter dialog comes up automatically every time you run a Where Used query.
You can also open that dialog later from the results panel to refine the search by clicking on the filter button

after you have obtained initial results.
The following graphic shows the dialog:

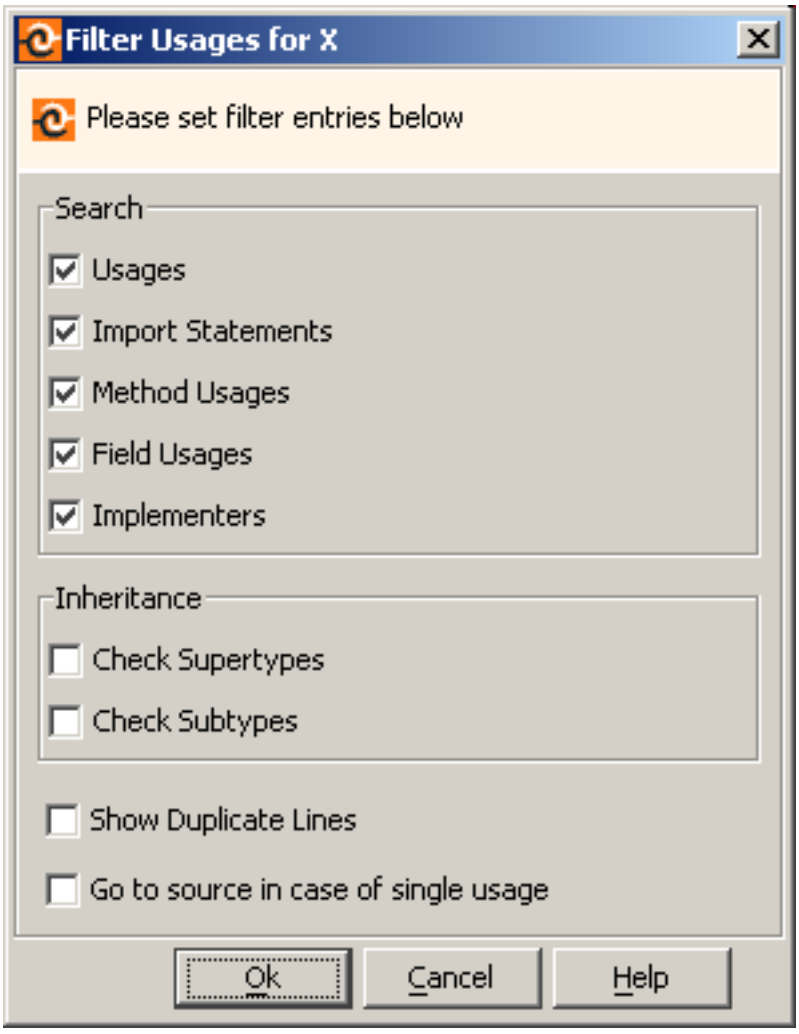


Name	Description
Usages	Search for usages of the class
Import Statements	Search for import statements that import the class
Field Usages	Search for places where a field of the class is used
Check Subtypes/Check Supertypes	Report usages that are invoked on subtypes/supertypes of the class that the method being searched belongs to; for more information and an example, see Supertypes & Subtypes.
Show Duplicate Lines	If turned off, multiple usages on the same line will be merged into one line in the results panel.
Go to source in case of single usage	If checked and only one usage is found, the result window will not be displayed. Instead, the user is taken to the usage in the source file.

Filter Dialog for Interfaces

You can narrow down the results of a Where Used query by choosing different settings rom the query's filter dialog. The filter dialog comes up automatically every time you run a Where Used query.
You can also open that dialog later from the results panel to refine the search by clicking on the filter button

after you have obtained initial results.
The following graphic shows the dialog:



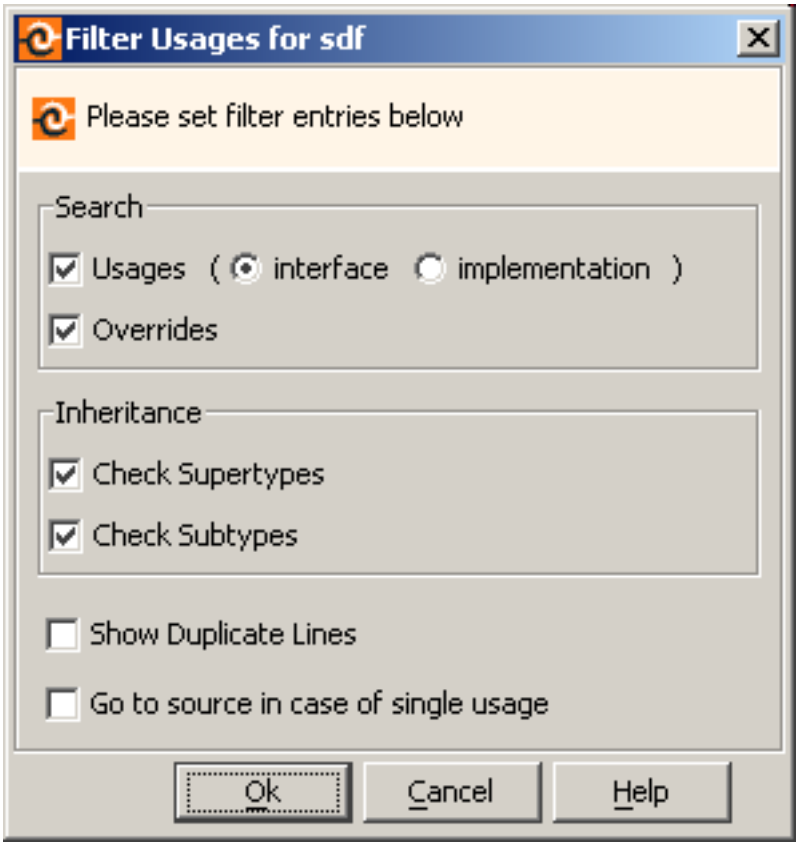
Name	Description
Usages	Searches for usages of the class
Import Statements	Searches for import statements that import the class

Field Usages	Searches for places where a field of the class is used
Implementers	Searches for classes that implement this interface
Check Subtypes/Check Supertypes	Reports usages that are invoked on subtypes/supertypes of the class that the method being searched belongs to; for more information and an example, see Supertypes & Subtypes.
Show Duplicate Lines	If turned off then multiple usages on the same line will be merged into one line in the results panel.
Go to source in case of single usage	If checked and only one usage is found, the result window will not be displayed. Instead, the user is taken to the usage in the source file.

Filter Dialog for Non-Private Methods

You can narrow down the results of a Where Used query by choosing different settings rom the query's filter dialog. The filter dialog comes up automatically every time you run a Where Used query.
You can also open that dialog later from the results panel to refine the search by clicking on the filter button

after you have obtained initial results.
The following graphic shows the dialog:



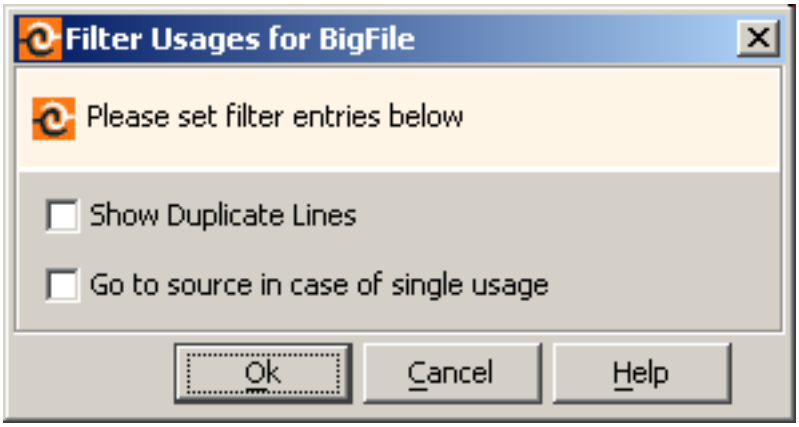
Name	Description
Interface usages vs. implementation usages	The distinction between "interface usages" and "implementation usages" matters only when you are using "check supertypes" and/or the "check subtypes" option. Interface usage means using any method in subclasses/superclasses hierarchy that has exactly the same signature as the target method (for instance, using overriding methods counts as interface usage). Implementation usage means using a particular method body--so, overrides don't count, but simply using the method through a subclass's method does (note that this would also count as interface usage).
Overrides	If checked, RefactorIT also includes all overriding methods in the result.

Check Subtypes/Check Supertypes	Reports usages that are invoked on subtypes/supertypes of the class that the method being searched belongs to; for more information and an example, see Supertypes & Subtypes.
Show Duplicate Lines	If turned off then multiple usages on the same line will be merged into one line in the results panel.
Go to source in case of single usage	If checked and only one usage is found then the result window will not be displayed -- instead, the user is taken to the usage in the source file.

Filter Dialog for Constructors and Private Methods

You can narrow down the results of a Where Used query by choosing different settings rom the query's filter dialog. The filter dialog comes up automatically every time you run a Where Used query.
You can also open that dialog later from the results panel to refine the search by clicking on the filter button

after you have obtained initial results.
The following graphic shows the dialog:



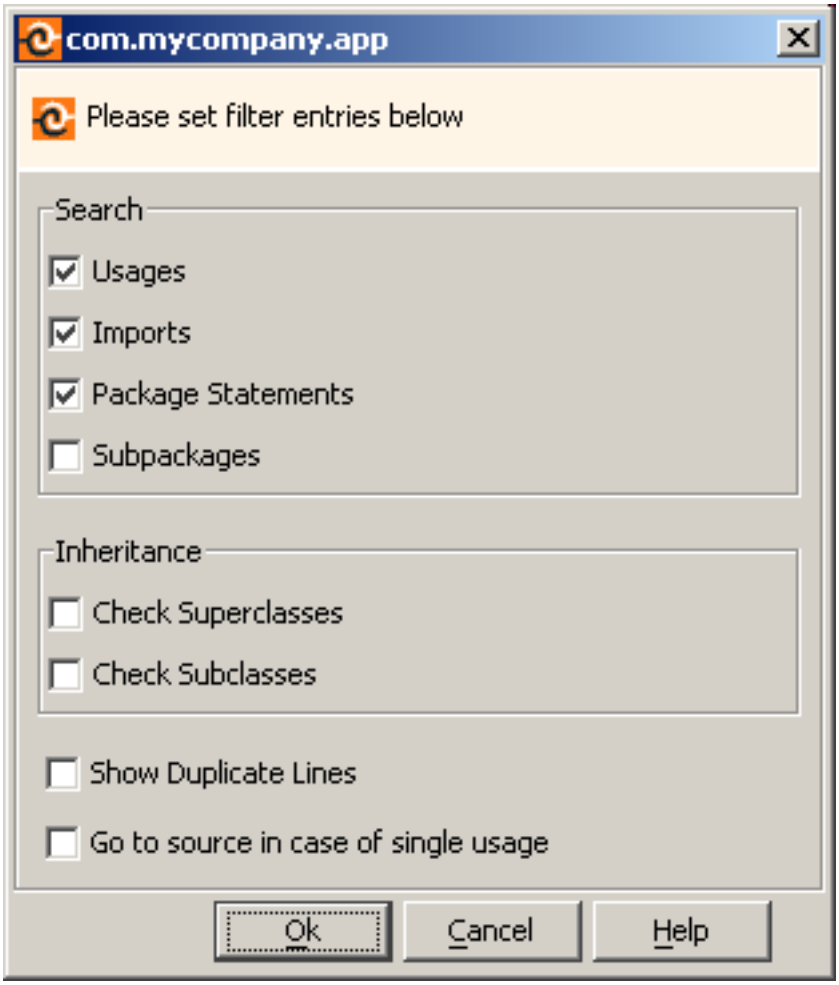
Name	Description
Show Duplicate Lines	If turned off, multiple usages on the same line will be merged into one line in the results panel.

Go to source in case of single usage	If checked and only one usage is found, the result window will not be displayed. Instead, the user is taken to the usage in the source file.

Filter Dialog for Packages

You can narrow down the results of a Where Used query by choosing different settings rom the query's filter dialog. The filter dialog comes up automatically every time you run a Where Used query.
You can also open that dialog later from the results panel to refine the search by clicking on the filter button

after you have obtained initial results.
The following graphic shows the dialog:

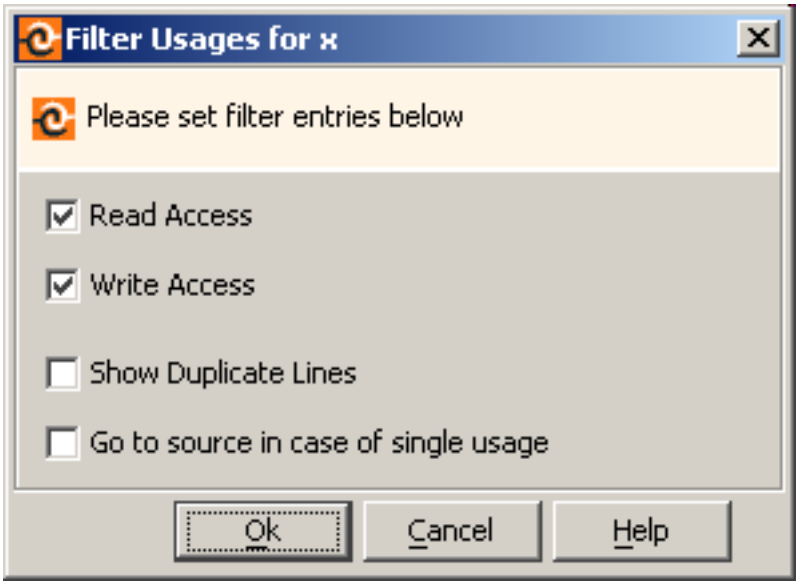


Name	Description
Usages	Search for usages of the class
Import Statements	Search for import statements that import the package
Package Statements	Search for package declaration statements
Subpackages	Also search for subpackages in the search
Check Subtypes/Check Supertypes	Report usages that are invoked on subtypes/supertypes of the class that the method being searched belongs to; for more information and an example, see Supertypes & Subtypes.
Show Duplicate Lines	If turned off then multiple usages on the same line will be merged into one line in the results panel.
Go to source in case of single usage	If checked and only one usage is found, the result window will not be displayed. Instead, the user is taken to the usage in the source file.

Filter Dialog for Variables

You can narrow down the results of a Where Used query by choosing different settings rom the query's filter dialog. The filter dialog comes up automatically every time you run a Where Used query.
You can also open that dialog later from the results panel to refine the search by clicking on the filter button

after you have obtained initial results.
The following graphic shows the dialog:



Name	Description
Read Access	Search for places where the variable is read
Write Access	Search for places where the variable is written
Show Duplicate Lines	If turned off, multiple usages on the same line will be merged into one line in the results panel.
Go to source in case of single usage	If checked and only one usage is found, the result window will not be displayed. Instead, the user is taken to the usage in the source file.

Supertypes and Subtypes

For most Where Used queries, you can specify that RefactorIT should also look for usages that are invoked on supertypes/subtypes of the type that the item being searched belongs to. These options are called "check supertypes" and "check subtypes".
Depending on the item being searched, the filters are initially turned on or off to yield the most expected result.
When checking for supertypes, RefactorIT also checks for supertypes of supertypes and so on until the top of inheritance hierarchy (java.lang.Object) is reached. Similarily, subtypes are also checked in the entire inheritance hierarchy.

Example

Usages of B.test() are searched in the code below:

```
class A {
  void test() {}
}

class B extends A {
  void test() {}
}

class C extends B {}

class Test {
  void test() {
    A a = new C();
    a.test();
    B b = new C();
    b.test();
    C c = new C();
    c.test();
  }
}
```

Command Line Interface

The Audit, Metrics and Not Used operations can be ran from the command line and by a custom Ant task.
Results can be saved to files in various formats, so they can be analyzed by spreadsheet tools, or, for example, sent via e-mail to the develpoment team as part of the nightly build.
Results with too much "noise" will tend to be skipped by people, which is why Audit and Metrics results are optionally configurable using profiles.

Command Line Switches

Usage: run.sh -nogui <options>
Where options include (some are optional):

-sourcepath <path>	Project sourcepath (default='.')
-classpath <path>	Project classpath (default='.')

-license <path>	License (optional for <50 sources)
-format <format>	Output format ('html','text' or 'comma-separated'; default='text')
-profile <path>	Audit or Metrics profile (opt)
-output <path>	Output file path (default=stdout)
-metrics	Runs Metrics on the project
-notused	Runs NotUsed on the project
-audit	Runs Audit on the project

Example 1: run Metrics under Windows:

run.bat -nogui -sourcepath c:\temp -classpath c:\temp\x.jar -metrics -format comma-separated
Example 2: run Audits under Windows with a specific profile and a license file:

run.bat -nogui -sourcepath c:\temp -audit -profile c:\temp\my_profile.profile -license c:\temp\refactorit.license
Command Line Overview

RefactorIT Ant task

Description

Runs RefactorIT actions, such as audit, metrics and not used. For a quick overview, see a working sample file.

Installation

Just install RefactorIT in standalone mode (it's an installer option). In build files, use the following task definition (be sure to replace "C:\RefactorIT" with your RefactorIT home folder):

```
<taskdef name="RefactorIT" classname="com.xpdev.refactory.ant.RefactorIt">
  <classpath>
    <fileset dir="C:\RefactorIT" includes="**/*.jar"/>
  </classpath>
</taskdef>
```

Parameters

Attribute	Description	Required
action	Action to run: "metrics", "audit" or "notused"	Yes
sourcepath	Sourcepath used for loading the project; could be specified via nested "sourcepath" element(s), as a path-like structure	Yes, unless provided in another way
classpath	Classpath used for loading the project; could be specified via nested "classpath" element(s), as a path-like structure	Yes, unless provided in another way
sourcepathref	Sourcepath, given as a reference	No
classpathref	Classpath, given as a reference	No
license	RefactorIT license file location	Yes, unless use

		d with a small project
format	text, html or comma separated	No
output	Output file name	No
profile	Profile for metrics or audits	No

Examples

See a working sample file.

Profiles for Audit and Metrics

Audit and Metrics results can be fine-tuned via loadable and saveable profiles which can also be used to filter results for the command line, using the "profile" switch or Ant-task paramerter.
To edit, save or load these profiles, just run your chosen action (Audit or Metrics) on some item. This brings up the profile dialog for that action.

Frequently Asked Questions

- Memory
- Sourcepath

Memory

How to configure memory allocation for SunTM ONE Studio

-J-Xverify:none -J-Xms24m -J-Xmx96m

For example, you can modify the value of the -J-Xmx parameter to be 128 Mb like this:

-J-Xverify:none -J-Xms24m -J-Xmx128m

Note: It is advisable to set the startup memory (-J-Xms) to the amount that SunTM ONE Studio will probably need. Programs tend to run faster when they have enough memory.
Memory configuration is saved in the SunTM ONE Studio file bin/ide.cfg. The entry looks like this by default:

How to increase memory for the standalone version

lax.nl.java.option.java.heap.size.initial=128M
lax.nl.java.option.java.heap.size.max=256M

How to use part of a project for refactoring

Then you may wish to set the subdirectory you are working on as a "source path" for RefactorIT. In order to satisfy dependencies, RefactorIT needs to know the ".class" files of the rest of your project. This is done by adding the directory containing these classes to the "classpath". It is OK if this directory also contains your current development classes. Follow the step by step instructions:

It may happen that you are developing only on certain parts of a project, especially if it is a big project.
Standalone version

1. Go to the Options dialog box (Project->Options) or, create a New Project.
   Hint: use double-click to edit
2. Set the source sub-directory you want to work with as the "sourcepath". This should be one entry only.
3. Set the directory containing all compiled classes as the "classpath" and add other required libraries also.

4. Click OK and select File->Save to save the new project settings.
5. Reload the project (File->Open).

SunTM ONE Studio / NetBeans version

1. Go to the RefactorIT Options dialog box (Tools->RefactorIT->RefactorIT Options).
2. Enable "Specify classpath and sourcepath".
3. Set your source sub-directory you want to work with as the "sourcepath". This should be one entry only.
4. Set the directory containing all compiled classes as the "classpath" and add other required libraries also.
5. Restart the IDE to be sure the changes take effect.

JDeveloper9i

1. Go to the Project Settings.
2. Set your source sub-directory you want to work with in Common/Input Paths/Java Source Path.
3. Add the directory containing all compiled classes to "Configurations/Development/Libraries/Selected libraries".
4. Click OK to save the changes.

RefactorIT Errors

If RefactorIT shows parsing or resolving errors on project errors tab and you are sure that your code compiles correctly you should check:

- Is project sourcepath correct?
- Does project classpath contain all imports you need?

If you are sure that both these are correct you can

1. Choose **Rebuild** or **Clean** command from RefactorIT menu. This forces RefactorIT to rebuild project.
2. Restart IDE

If neither of them helps, please contact RefactorIT Support at support@refactorit.com.

See Also

Project Options
Cache